OVERVIEW OF MULTIDATABASE TRANSACTION MANAGEMENT

Yuri Breitbart
Hector Garcia-Molina
Avi Silberschatz

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

TR-92-21 May 1992

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712

Overview of Multidatabase Transaction Management

Yuri Breitbart[1]
Department of Computer Science
University of Kentucky
Lexington, KY 40506

Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA 94305

Avi Silberschatz[2]
Department of Computer Sciences
University of Texas
Austin, TX 78712

Abstract

A multidatabase system (MDBS) is a facility that allows users access to data located in multiple autonomous database management systems (DBMSs). In such a system, *global transactions* are executed under the control of the MDBS. Independently, *local transactions* are executed under the control of the local DBMSs. Each local DBMS integrated by the MDBS may employ a different transaction management scheme. In addition, each local DBMS has complete control over all transactions (global and local) executing at its site, including the ability to abort at any point any of the transactions executing at its site. Typically, no design or internal DBMS structure changes are allowed in order to accommodate the MDBS. Furthermore, the local DBMSs may not be aware of each other, and, as a consequence, cannot coordinate their actions. Thus, traditional techniques for ensuring transaction atomicity and consistency in homogeneous distributed database systems may not be appropriate for an MDBS environment. The objective of this paper is to provide a brief review of the most current work in the area of multidatabase transaction management. We first define the problem and argue that the multidatabase research will become increasingly important in the coming years. We then outline basic research issues in multidatabase transaction management and review recent results in the area. We conclude the paper with a discussion of open problems and practical implications of this research.

[1]This material is based in part upon work supported by the Center for Manufacturing and Robotics of the University of Kentucky and by NSF Grant IRI-8904932.

[2]This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant Nos. IRI-9003341 and IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

1 Introduction

Recent progress in communication and database technologies has changed the user data processing environment. The present data processing situation is characterized by a growing number of applications that require access to various *pre-existing* local data sources located in *heterogeneous* hardware and software environments distributed among the nodes of a network. Each local data source is a collection of data and applications that are run under a particular database management system (DBMS) and are administered/operated under a particular policy or local rules.

The data sources are *pre-existing* in the sense that they were created independently, in an uncoordinated way and without considering that one day they may need to be integrated. The DBMSs involved are *heterogeneous* in the sense that they operate in different environments and may use different underlying data models, data definition and data manipulation facilities, transaction management and concurrency control mechanisms, and physical data structures.

A *multidatabase* is composed of local data sources. Systems that facilitate the logical integration of local data sources are called *multidatabase systems*. Logical data integration creates an illusion of a single database system and hides from users the intricacies of different DBMSs and different access methods. It provides users with uniform access to data contained in various databases, without migrating the data to a new database, and without requiring the users to know either the location or the characteristics of different databases and their corresponding DBMSs. Using the multidatabase approach, pre-existing applications remain operational and new applications may access data in various distributed data sources.

A multidatabase system (MDBS) is built on top of a number of local DBMSs that manage different local data sources. Access to data located in a local data source is accomplished through *transactions*. A transaction results from the execution of a user program written in a high level programming language (e.g., *C*, or *PASCAL*). In this paper, we assume that each local DBMS ensures the following properties (called *ACID* properties) of transactions executed at its site

- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation:** Each transaction assumes that it is executed alone in the system and the local DBMS guarantees that intermediate transaction results are hidden from other concurrently executed transactions.
- **Durability:** The values changed by the transaction must persist after the transaction successfully completes.

To ensure the consistency and isolation properties, each local DBMS generates a conflict serializable schedule consisting of operations of local and global transactions that were executed at its site. To ensure the atomicity and durability properties, each local DBMS uses some form of recovery scheme (e.g., write-ahead log scheme [7]).

The MDBS considers each local DBMS as a *black box* that operates *autonomously*, without the knowledge of either other local DBMSs or the MDBS system. Local autonomy is the main feature that distinguishes the multidatabase systems from conventional distributed database systems. There are three main types of autonomy:

- **Design autonomy:** No changes can be made to the local DBMS software to accommodate the MDBS system. Making changes to the existing software of the DBMS is expensive, may result in performance degradation, and, further, may render pre-existing applications inoperative.

- **Execution autonomy:** Each local DBMS should retain complete control over the execution of transactions at its site. An implication of this constraint is that a local DBMS may abort a transaction executing at its site at any time during its execution, including the time when a global transaction is in the process of being committed by the MDBS.

- **Communication autonomy:** Local DBMSs integrated by the MDBS are not able to coordinate the actions of global transactions executing at several sites. This constraint implies that local DBMSs do not share their control information with each other or with the MDBS system.

Participating DBMSs may have different autonomy levels. For example, some sites may be willing to participate in the coordination of a global transaction (low communication autonomy) while others may not (high communication autonomy).

One way to characterize the autonomy levels of sites is to define the *interface* that each local data source offers to user transactions. For example, no airline, or bank, or car agency would allow to external users transactions to access their data using SQL statements. On the other hand, internal users transactions will be allowed to do so. The interfaces can be categorized by the *operations* they accept from the MDBS. Here, we illustrate some of the operations that may be available at a site (black box). We partition these operations into two sets. The first one deals with transaction operations, while the second one deals with status information.

- Transaction operations:
  - **Begin transaction**. The MDBS requests that a new local transaction be initiated. DBMS typically returns a transaction identification to be used in later commands.
  - **End transaction**. The identified transaction has completed and may be committed.
  - **Read** or **Write**. Perform indicated action. The action may be low level (e.g., read a record or write a record) or high level (e.g., withdraw money from an account). Begin and end transaction operations may be implicit in action.
  - **Abort**. Terminate and abort a transaction. Undo all transactions effects in the database.
  - **Commit**. Make all changes that a transaction made permanent in the database and purge the transaction from the system
  - **Prepare to Commit.** The identified transaction has finished its actions and is ready to commit. DBMS guarantees that transaction will not be unilaterally aborted and waits for commit or abort decision from the MDBS.
  - **Service Request.** The execution of a procedure is requested (e.g., "reserve a seat on a given flight"). A service request is equivalent to submitting all the actions of a local transaction, from begin transaction to commit, at once.

- Status information operations:
  - **Get-wait-for-graph.** Retrieve the local-wait-for-graph (if one is used) to be used in global deadlock detection. (A local-wait-for-graph consists of a set of vertices corresponding to transaction names, and a set of edges specifying a waiting relation between transactions. A cycle in the graph indicates a deadlock situation.)

– **Get-serialization-order.** Retrieve information regarding the commit order of transactions. (Such an order can be represented by a serialization graph, where the sets of vertices correspond to transaction names, and the set of edges specify serialization order. A cycle in the serialization graph indicates a non-serializable schedule.)

– **Inquire.** Find out status (e.g., commit, abort) of a transaction.

– **Disable transaction class.** Certain types of transactions (e.g., identified by semantic type, or by read or write access sets) are not allowed to commit at this box.

Thus, each local data source *exports* a well defined set of high-level operations that may be invoked by users transactions. This notion of exported high-level operations roughly corresponds to the transactions *steps* [22, 65]. In addition to the available operations, the global system may also use *knowledge* of the internals of the local DBMS. For example, it may be known that a local DBMS uses the two phase locking (2PL) protocol [18], or that it uses a strict, recoverable or cascadeless concurrency control mechanism [7]. As we will see, this information may be of use to the MDBS for coordinating global transactions.

The operations define a spectrum of autonomy. At one end we have sites that simply provide a service request interface (see choice (g) above); the MDBS is offered a fixed choice of services, and once a request is submitted the MDBS has no control as to when it is executed, if at all. At the other end of the spectrum the various transaction operations are submitted individually to the local DBMS. Thus, the only mechanism that the MDBS can use to guarantee certain properties of transaction executions is the mechanism of coordinating of a submission of transaction's operations.

A variety of points in the spectrum have been studied in the literature. In general, the more autonomy the DBMSs retain, the harder it is to guarantee global data consistency [28, 29]. In this paper we provide an overview of the different points of the spectrum that have been studied and the corresponding MDBS transaction mechanisms. Since our focus in this paper is on high level autonomy, we will not consider here multidatabase systems where the participating local DBMSs export *wait-for* or *serialization-order* information [53, 57, 51]. Such MDBSs are closely related to homogeneous distributed database systems that have been extensively studied [7]. The results in this paper are also applicable to homogeneous distributed database systems that allow local sites to retain some level of execution autonomy.

We start by discussing our multidatabase transaction management model, and analyzing the problems that arise in a multidatabase environment. We then classify different notions of global database consistency that have been introduced in literature so far. Finally, we discuss some open problems that still need to be solved. This review is not intended to be comprehensive, but hopefully covers major progress to date.

2 Multidatabase Transaction Model

In this section we define the *base* transaction model to be used throughout the paper. This model is chosen because it is the one that has received the most attention in the multidatabase literature. However, we will consider extensions to the basic model at later points. We assume that each local DBMS interface includes at least read, write, commit, and abort operations.

An MDBS consists of a number of pre-existing and autonomous local DBMSs located at sites s_1, s_2, ..., s_m, where $m \geq 2$. A transaction T_i is a sequence of *read* (r_i) and *write* (w_i) operations terminated by either a *commit* (c_i) or an *abort* (a_i) operation. A multidatabase environment supports two types of transactions:

- **local transactions**, those transactions that access data managed by only a single DBMS. These transactions are executed by the local DBMS, outside of MDBS control
- **global transactions**, those transactions that are executed under MDBS control. A global transaction consists of a number of subtransactions, each of which is an ordinary local transaction from the point of view of local DBMS where the subtransaction is executed.

The local schedule at site s_k, denoted by S_k, is a sequence of local and global transactions operations resulting from their execution at site s_k. Transaction T_i is said to be *committed* (*aborted*) in S_k if S_k contains c_i (a_i) operation. Transaction T_i is active in S_k if it is neither committed nor aborted in S_k. A projection of S_k on a set of transactions T is a schedule that contains only operations of transactions from T. A committed projection of schedule S_k is a schedule that contains only operations of committed transactions in S_k.

We say that transactions T_i and T_j are in *direct conflict* in schedule S_k if and only if schedule S_k contains operation $o_i(x)$ followed by operation $o_j(x)$, where $o_i(x)$ or $o_j(x)$ are a write operation and T_i does not abort before $o_j(x)$ is executed. We say that transactions T_i and T_j are in *indirect conflict* in schedule S_k if and only if there is a sequence of transactions $T_1, T_2, \ldots, T_r$ such that T_i is in direct conflict with T_1, T_1 is in direct conflict with T_2, ..., and, finally, T_r is in direct conflict with T_j. Transactions T_i and T_j are in conflict if and only if they are in direct or indirect conflict.

Two local schedules are equivalent if they are defined on the same set of global and local transactions, have the same operations and the same set of pairs of conflicting committed transactions. Schedule S_k is conflict serializable if it is equivalent to a serial schedule. A local serialization graph for schedule S_k is a directed graph with nodes corresponding to global and local transactions that are committed in S_k and with a set of edges such that $T_i \rightarrow T_j$ if T_i conflicts with T_j in S_k. Schedule S_k is serializable if and only if its local serialization graph is acyclic [7].

A global schedule S is a partial ordered set of all operations belonging to local and global transactions such that, for any local site s_k, a projection of S on a set of global and local transactions executing at site s_k is the local schedule S_k at site s_k. We say that a global schedule is *globally serializable* if and only if there exists a total order defined over committed global transactions that is consistent with the serialization order of committed global transactions at each of the local DBMSs [9]. An union of local serialization graphs is called a global serialization graph. A global schedule is globally serializable if and only if its global serialization graph is acyclic [9].

The MDBS software that executes on top of the existing local DBMSs consists of a *global transaction manager* (GTM), and a set of *servers*, one associated with each local DBMS. Each global transaction submits its operations to the GTM. For each submitted operation, the GTM determines whether to submit the operation to local sites, or to delay it, or to abort the transaction. If the operation is to be submitted, the GTM selects a local site (or a set of sites) where the operation should be executed.

The GTM submits global transaction operations to the local DBMSs through the server which acts as the liaison between the GTM and the local DBMS. Operations belonging to a global subtransaction are submitted to the local DBMS by the server as a single transaction. We assume that each local DBMS acknowledges to the server (and, in turn, to the GTM) the execution of operations submitted to it. We do assume that the actions of a given transaction at a site always end an execution with a commit (or abort) operation.

We do not impose any restrictions on how the various *read* and *write* operations of a global transaction are executed by the GTM. It is possible in our model for several operations of the same transaction to be executed by the GTM at the same time (parallel execution) or for no operation

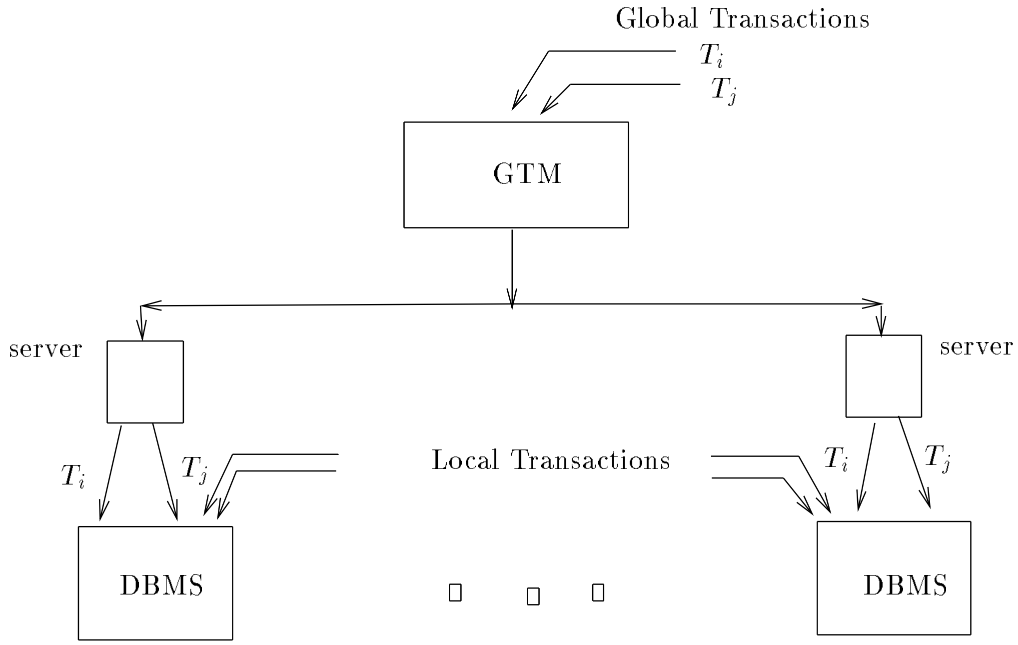

Figure 1: The MDBS Model

of the transaction (except the very first one) to be submitted for execution until the GTM receives an acknowledgment from the previous operation of the same transaction (serial execution). The overall multidatabase system model is depicted in Figure 1.

As mentioned earlier, we will also consider variations of base model we have presented. In particular, we will consider two variations:

- **Service Interface**. Many real life examples of multidatabase applications are based on a high-level service interface model [31, 20] (e.g. networks of travel agencies, the international interbank clearing system, etc.). In the service interface model the GTM submits service requests as opposed to individual read, write, abort, and commit actions. A service request generates read, write, commit (or abort) at local sites, just like in the base model. However, the GTM receives a single acknowledgment, after all actions have committed (or aborted). In most cases, global concurrency control mechanisms are not significantly affected if one assumes a service interface model as opposed to the base model. Where there is some impact, we will point it out.

- **Extended Base Model**. At various points, we will assume that sites provide additional operations, such as a prepare-to-commit one. We will also assume several types of knowledge about the concurrency control mechanisms used at local sites. For each extension, we will study how global transaction management is affected.

3 Multidatabase Transaction Management Issues

The Global Transaction Manager (GTM) should guarantee the ACID properties of global transactions, even in the presence of local transactions that the GTM is not aware of. In addition, the GTM should guarantee deadlock free executions of global transactions and it should provide means to recover from any type of system failures. In the next three subsections we illustrate the difficulties that may arise.

3.1 Global Serializability Problem

The various local DBMSs may use different concurrency control protocols (e.g., *Two Phase Locking* (2PL), *Timestamp Ordering* (TO), *Serialization Graph Testing* (SGT), etc.). Existing solutions for ensuring global serializability in a homogeneous distributed database assume that each site uses the same concurrency control scheme and shares its control information; hence existing solutions cannot be used in a MDBS environment.

Since local transactions execute outside the control of the GTM, the GTM can guarantee global serializability only through the control of the execution order of global transactions. However, in such an environment, even a serial execution of global transactions does not guarantee global serializability. The following example illustrates this fact.

Example 3.1: Consider a multidatabase system located at two sites: s_1 with data items a and b, and s_2 with data items c and d. Let T_1 and T_2 be two read-only global transactions defined as follows:

$$\begin{array}{ll} T_1: & r_1(a)\; r_1(c) \\ T_2: & r_2(b)\; r_2(d) \end{array}$$

In addition, let T_3 and T_4 be two local transactions at sites s_1 and s_2, respectively, defined as follows:

$$\begin{array}{ll} T_3: & w_3(a)\; w_3(b) \\ T_4: & w_4(c)\; w_4(d) \end{array}$$

Assume that transaction T_1 is executed and committed at both sites and after that transaction T_2 is executed and committed at both sites. Such execution may result in the following local schedules S_1 and S_2 generated at sites s_1 and s_2, respectively:

$$\begin{array}{ll} S_1: & r_1(a)\; c_1\; w_3(a)\; w_3(b)\; c_3\; r_2(b)\; c_2 \\ S_2: & w_4(c)\; r_1(c)\; c_1\; r_2(d)\; c_2\; w_4(d)\; c_4 \end{array}$$

As a result, transaction T_1 is serialized before T_2 at s_1 and after T_2 at s_2; hence global serializability is not maintained. □

In Example 3.1, the problem arises because the local transactions create *indirect* conflicts between global transactions. Since the GTM is not aware of local transactions, it is also not aware of these indirect conflicts. This phenomena is a cause of major difficulties in trying to ensure global serializability in a multidatabase environment.

3.2 Global Atomicity and Recovery Problems

The global atomicity requirement dictates that either all the subtransactions of a transaction commit, or they all abort. In a homogeneous distributed database systems, atomicity of transactions is ensured by an *atomic commit protocol* [7]. This protocol requires that the participating local sites provide a*prepared state* for each subtransaction. The subtransaction should remain in the *prepared* state until the coordinator decides whether to commit or abort the transaction.

If we wish to preserve the execution autonomy of each of the participating local DBMSs, then we must assume that local DBMSs do not export a transaction's prepared state. In such an environment, a DBMS can unilaterally abort a subtransaction any time before its commit. This not only leads to global transactions that are not atomic, but to incorrect global schedules, as illustrated below.

Example 3.2: Consider a global database consisting of two sites s_1 with data item a, and s_2 with data item c. Consider the following global transaction T_1:

$$T_1\colon\ r_1(a)\ w_1(a)\ w_1(c)$$

Suppose that T_1 has completed its read/write actions at both sites and the GTM sends commit requests to both sites. Site s_2 receives the commit and commits its subtransaction. However, site s_1 decides to abort its subtransaction before the commit arrives. Therefore, at site s_1 the local DBMS undoes the T_1 actions. After this is accomplished a local transaction T_2:

$$T_2\colon\ r_2(a)\ w_2(a)$$

is executed and committed at the site.

At this point, the resulting global schedule is incorrect, as it only reflects the s_2 half of T_1. To correct the situation, say the GTM attempts to redo the missing actions by resubmitting to s_1 the missing write $w_1(a)$. The local DBMS, however, considers this operation as a new transaction T_3 that is not related to T_1. Thus, from the local DBMS viewpoint, the committed projection of the s_1 schedule is:

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

However, T_3's write operation is the same as $w_1(a)$ as far as the MDBS is concerned. Thus, this execution results in the following non-serializable schedule S_1:

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

We note that if the local DBMSs provide a prepare-to-commit operation, and they participate in the execution of a global commit protocol, then the problems shown in Example 3.2 can be avoided. In particular, in that example, the GTM does not issue the commit actions for T_1 until both sites have acknowledged the prepare-to-commit. Because s_1 is prepared for T_1, it cannot abort it and the situation shown in Example 3.2 does not arise. However, as discussed above, this will violate the execution autonomy requirement.

There is an ongoing debate as to whether sites in a MDBS will provide prepare-to-commit operations and thereby give up their execution autonomy. One side argues that the two phase commit protocol (with the prepared-to-commit operation) is becoming a standard, so that soon *all* DBMS will provide this service. The other side argues that there will always be autonomous sites that will want to preserve their execution autonomy, and therefore will not want to export the prepare-to-commit operation, even if their local DBMSs provide it. This is because they do not want their site to hold resources (e.g., locks) on behalf of a remote transaction, which may last for an indefinite amount of time. The first camp counter-argues that with modern networks and computers, global transactions will be very fast, so the time that a site needs to block its resources is minimal. So the site administrators will not mind allowing the prepare-to-commit. Furthermore, they claim, the operator at a site can always manually release a transaction that hangs for too long (e.g., break locks manually). So if a transaction ever waits too long in its prepare-to-commit state, it can be aborted. The second camp then counter-argues that if the prepare-to-commit commitment can be broken by the operator, then sites can unilaterally abort after all, so we are back at square one.

Without taking sides in the argument, we believe it is important to study both scenarios, with or without prepare-to-commit at the sites. In this paper, we will review both cases.

3.3 Global Deadlock Problem

Consider a multidatabase system where each local DBMS uses a locking mechanism to ensure local serializability. We assume that each local DBMS has a mechanism to detect and recover from local deadlocks. However, in such systems there is a possibility of a global deadlock that cannot be detected by the GTM.

Example 3.3: Consider a multidatabase system located at two sites: s_1 with data items a and b, and s_2 with data items c and d. Local DBMSs at both sites use the two phase locking protocols to guarantee local serializability. Let T_1 and T_2 be two global transactions defined as follows:

$$\begin{array}{ll} T_1: & r_1(a)\, r_1(d) \\ T_2: & r_2(c)\, r_2(b) \end{array}$$

In addition, let T_3 and T_4 be two local transactions at sites s_1 and s_2, respectively, defined as follows:

$$\begin{array}{ll} T_3: & w_3(b)\, w_3(a) \\ T_4: & w_4(d)\, w_4(c) \end{array}$$

Assume that T_1 has executed $r_1(a)$ and T_2 has executed $r_2(c)$. After that at site s_1 local transaction executes $w_3(b)$, submits $w_3(a)$ and is forced to wait for a lock on a that is kept by T_1. At site s_2, transaction T_4 executes $w_4(d)$, submits $w_4(c)$ and is forced to wait for a lock on c that is kept by T_2. Finally, transactions T_1 and T_2 submit their last operations and a global deadlock ensues. □

Due to the design autonomy, local DBMSs may not wish to exchange their control information and therefore will be unaware of the global deadlock. Similarly, the MDBS is not aware of local transactions and, therefore, will be also unaware of the deadlock. In Section 7 we will discuss what the GTM can do to ensure deadlock freedom.

4 Global Serializability Schemes

In this section, we describe techniques for ensuring global serializability in a failure and abort free environment. That is, we assume that each transaction, whether local or global, will always successfully complete once it has been submitted for execution. Under this assumption, no aborts of global transactions by local DBMSs due to the local deadlocks are permitted. This is clearly not a realistic assumption. However, studying this simplified scenario yields a formal understanding of the synchronization issues that arise in dealing with independent concurrency control mechanisms. Failures and aborts will be considered in Section 6.

Example 3.1 demonstrates the key problem in guaranteeing globally serializable schedules. Local transactions (such as T_3) may generate indirect conflicts between global transactions that otherwise are not in conflict. In the example, T_3 creates a dependency $T_1 \rightarrow T_3 \rightarrow T_2$ and at the second site T_4 creates dependencies $T_2 \rightarrow T_4 \rightarrow T_1$. Thus, even though T_1 and T_2 do not conflict, they are involved in a cycle in the global serialization graph.

To avoid these cycles, the GTM will have to take some action. What action is taken depends on the amount of knowledge the GTM has concerning the local concurrency control mechanisms. In the subsections that follow we consider various scenarios, and for each one explain the types of GTM actions that will ensure global serializability. The base scenario (Section 4.1) corresponds to our base transaction model (Section 2): the GTM simply knows that each local site generates local serializable and deadlock free schedules. Thus, the GTM considers each such DBMS as an *unlabeled black box*. In subsequent scenarios, the GTM assumes additional properties (labels) of the sites (black boxes).

In general terms, the actions taken by the GTM can be of two types:

- **Pessimistic**. Global transactions are delayed to avoid serialization graph cycles.
- **Optimistic**. Cycles or potential cycles are detected and broken by aborting global transactions.

There is a tradeoff between these two approaches: a pessimistic approach does not generate transaction aborts but may result in lower concurrency, while an optimistic approach may increase concurrency but may result in a large number of transactions aborts [14].

4.1 Integration with Unlabeled DBMSs

Continuing with Example 3.1, consider a snapshot of schedules S_1 and S_2 depicting a situation where the GTM has executed T_1 but has not started T_2. (The actions that have *not* been executed yet are within brackets.)

$$S_1\text{: } r_1(a)\ c_1\ [\ w_3(a)\ w_3(b)\ c_3\ r_2(b)\ c_2\ ]$$
$$S_2\text{: } w_4(c)\ r_1(c)\ c_1\ [\ r_2(d)\ c_2\ w_4(d)\ c_4\ ]$$

The GTM wishes to avoid the future execution shown in the brackets, as it will result in a non-serializable schedule. One option would be to delay the execution of T_2 until the GTM is certain that a serializability cycle cannot occur. Unfortunately, since the GTM has no control over site s_2, it has no way of knowing when T_4 will complete, and T_2 needs to run after T_4 to avoid the dependency $T_2 \rightarrow T_4$. Hence, the pessimistic approach does not work. The optimistic one does not work either: when T_2 completes it would always have to be aborted since it could have participated in a cycle.

The only practical solution that is known to work in this scenario involves forcing conflicts among the global transactions [32]. In the example, we can force T_1 to write some object at every site it accesses data, and T_2 to read those objects. Thus, if the GTM executes T_2 after T_1 completes, then it ensures that the arc $T_1 \rightarrow T_2$ is placed in the global serialization graph. This guarantees that the arc $T_2 \rightarrow T_4 \rightarrow T_1$ cannot be generated at s_2, as it would create a *local* cycle. (Remember: the local site generates locally serializable schedules.) In the example, when T_4 submits its $w_4(d)$ action at s_2, the local cycle would be detected at T_4 would be aborted.

In the above example, global serializability was assured since T_2 ran after T_1 completed. However, if these two transactions were to run concurrently, then an additional mechanism is required to ensure that one site does not generate the edge $T_1 \rightarrow T_2$ while another site generates $T_2 \rightarrow T_1$ in their respective local serialization graphs. This is achieved by the use of a special data item-a *ticket* that is maintained at each local site. Only a single ticket is required for each local site, but tickets at different local sites are different data items. Only global transactions can access the ticket. Moreover, each global transaction executing at a site is required to read the ticket value, increment it and write an incremented value into the database. Thus, the ticket value read indicates the serialization order of the global transactions at the site.

The algorithm of [32] is optimistic: the GTM keeps a serialization graph for all active transactions (started but not committed). When a transaction T reads ticket value t at site s_i, an arc is entered from every transaction that read a ticket less than t at s_i to T. If T completes all of its actions and is not involved in a cycle, it is committed, else it is aborted. In [32] it is shown that the ticket method guarantees global serializability.

The ticket idea can also be used in a pessimistic way. In this case, global transactions are assigned a priori a global serialization order, and the tickets they should read are determined in advance. If a transaction submits its operation outside of a local site ticket order, it waits.

The performance of the ticket method has not been fully evaluated. It may lead to numerous aborted transactions (optimistic) or low concurrency (pessimistic). This same problem exists with most of the mechanisms we will survey later in this section, so it may be an inherent problem in trying to achieve global serializability with autonomous sites.

As a historical note, we mention that the paper [9] has proposed the *site-graph* mechanism as a way for ensuring global serializability in an environment where the local sites are assumed to be unlabeled black boxes. The proposed scheme, however, causes some transactions to be postponed indefinitely, unless the local sites tell the GTM when transactions such as T_4 (in our example) have completed. Unfortunately, this violates local autonomy. This site-graph mechanism is described later in Section 4.2.1.

4.2 Integration with Labeled DBMSs

We may be able to find global mechanisms that allow more concurrency, if we assume certain properties about the local sites. For example, suppose that the local sites use a basic timestamp ordering concurrency control algorithm. Returning to Example 3.1, say that execution has proceeded to the point indicated below (where the actions that have *not* been executed are again shown in brackets):

$$S_1\text{: } r_1(a)\ c_1\ w_3(a)\ w_3(b)\ c_3\ r_2(b)\ [\ c_2\ ]$$
$$S_2\text{: } w_4(c)\ r_1(c)\ c_1\ r_2(d)\ [\ c_2\ w_4(d)\ c_4\ ]$$

Consider now site s_2. Sometime during T_1's execution, it received a timestamp, say t_1. Sometime after T_2 starts, it will also receive a timestamp, t_2. Since T_1 and T_2 do not overlap in time, $t_2 > t_1$. The basic timestamp mechanism ensures that transactions are serialized in timestamp order, hence there can be no dependency $T_2 \rightarrow \cdots \rightarrow T_1$ in the local serialization graph. If T_4 were allowed to perform its second write action, it would create such a dependency; thus the local DBMS at s_2 will abort T_4.

This suggests a simple strategy for ensuring globally serializable schedules in an MDBS environment where each local DBMS is using a basic timestamp ordering concurrency control algorithm. The idea is to run global transactions serially. If transactions are not overlapped, we know that they will be assigned increasing timestamps, serialized in the proper order at each site, and the global schedule will be serializable. This simple scheme, however, forces global transactions to run serially. In the next three subsections we discuss MDBS systems that are able to concurrently execute global transactions by exploiting knowledge about properties of local schedules.

4.2.1 Strongly Serializable DBMSs

The following definition captures the essential property of timestamp ordering that lets us achieve global serializability. Actually, most concurrency control algorithms have this same property.

Definition 4.1: Let S be a serializable schedule. We say that schedule S is *strongly serializable* if and only if for every two transactions T_i and T_j in S, if the last operation of T_i (commit or abort) precedes the first operation of T_j, then there is some serial schedule equivalent to S where T_i precedes T_j (i.e., T_i precedes T_j in the S's serialization order). □

Assuming that a transaction receives a timestamp at the time of executing of its first operation, the basic timestamp ordering concurrency control algorithm ensures that a local schedule is strongly serializable.[3] Thus, as shown above, the GTM can ensure global serializability (when local schedules

[3]Without this assumption the basic timestamp ordering algorithm may generate not strongly serializable schedule. For example, $w_2(x)\ r_3(x)\ c_3\ w_1(y)\ c_1\ w_2(y)\ c_2$ can be generated by the basic timestamp ordering algorithm [7]. However, in this case the timestamp for T_1 was assigned before the first statement of T_1 was executed.

are strongly serializable) by executing global transactions serially. There are, however, several ways in which we can do better. For example, we note that if T_1 and T_2 execute at disjoint sites, there is no need to execute them serially. This suggests an algorithm where the GTM keeps a lock per site (the locks are kept at the GTM level, and not at the sites) [2]. Before a transaction can start, it must acquire the locks for all the sites it will run on. When it completes, the transaction releases its site locks. This ensures that transactions that could have generated a cycle like the one of Example 3.1 are run serially with respect to each other, thus avoiding the cycle due to the strong serializability of the local sites.

The lock-per-site approach is still overly restrictive. For example, consider two transactions: T_1 is to run at sites s_1 and s_2, and T_2 is to run at sites s_2 and s_3. If they run concurrently, a dependency may be generated at s_2, either $T_1 \rightarrow \cdots \rightarrow T_2$ or $T_2 \rightarrow \cdots \rightarrow T_1$. But since T_1 and T_2 do not interact at any other site, it does not matter: no global cycle can be generated in the global serialization graph.

This idea leads to essentially the site-graph algorithm of [9]. The GTM maintains a bipartite graph, with transactions and sites as the nodes. When a new transaction T_i is to be run, an arc is entered in the graph connecting the T_i node to each site node that T_i will run at. If there are no cycles, T_i runs. If a cycle involving T_i exists, then T_i is delayed until the cycle disappears.

To illustrate, say we try to run the schedule of Example 3.1. When T_1 starts, arcs (T_1, s_1) and (T_1, s_2) are entered. When T_2 starts, arcs (T_2, s_1) and (T_2, s_2) are entered, creating a cycle. Thus, T_2 is delayed until T_1 completes all of its actions. Since T_1 and T_2 are not overlapped, and since all local sites are strongly serializable, T_1 is serialized before T_2 at all sites. So a global cycle is avoided. If, on the other hand, T_2 runs only at sites s_2 and s_3, then there would be no cycle in the site graph and T_2 could run concurrently with T_1.

In the site graph algorithm, a transaction (node) cannot be removed from the graph upon the transaction's commit, if the transaction has a path in the graph that is connected to an uncommitted transaction. An aborted transaction, however, can be removed from the site graph as soon as it is aborted.

Another idea is to use altruistic locking to improve concurrency [2, 56]. To illustrate, say T_1 runs at sites s_1, s_2, and s_3, in that order. That is, T_1 first executes all of its actions at s_1 (including commit), then it executes at s_2, and then at s_3, with no overlapping of its actions at these sites. Suppose we run a second transaction, T_2, *in the wake* of T_1, that is, T_2 executes at s_1 after T_1 finished, and then runs at s_2 after T_1 finished there. Even though both T_1 and T_2 are executing concurrently, they are never overlapped at any one site, and T_2 always follows T_1. Hence, no site will generate a dependency $T_2 \rightarrow \cdots T_1$ and the global schedule will be serializable.

There are various ways to implement altruistic locking [2]. A simple way is to use site locks as before, except that transactions can release locks early if they know they have finished all processing at a site. However, the lock is not fully released; it is left in a "marked" state. Other transactions that request a site lock that is marked, can obtain the lock, but are then forced to be in the wake of the original transaction. The GTM must ensure that the relationship "is in the wake of" has no cycles. The latter can be done by keeping a *wake-graph* in which there is an edge between T_i and T_j if T_j is in the wake of T_i.

All the mechanisms we have described for strongly serializable local schedules are pessimistic. However, optimistic versions can easily be developed. For instance, in the site graph approach, instead of delaying transaction involved in a cycle, we could abort them.

4.2.2 Serialization-Point Based DBMSs

The notion of a strongly serializable schedule is closely related to that of a schedule that consists of transactions each of which has a *serialization point* [53]. A serialization point of a transaction is a distinguished action that determines the serialization order of the transaction in the schedule. For instance, in a concurrency control scheme based on timestamps, the distinguished action corresponds to the assignment of a timestamp to the transaction. When the transaction arrives and reads its timestamp, it will be serialized by the scheduler relative to other transactions according to its timestamp. That is, if two transactions contain conflicting operations and the scheduler serialized T_1 before T_2, then the timestamp of T_1 is smaller than the timestamp of T_2. Thus, a schedule that is generated by a timestamp scheduler consists of transactions that each one has a serialization point in the schedule.

In the 2PL scheme [18], the serialization point of a transaction corresponds to the operation of the first lock release. Once again, if transaction T_i contains a conflicting operation with T_j, and T_i is serialized before T_j, then T_i releases its first lock before T_j does. This leads us to introduce the following definition.

Definition 4.2: Let S be a serializable schedule consisting of transactions $T_1, T_2, \ldots, T_n$. We say that schedule S is an *sp-schedule* if and only if there exist a mapping *sp* from transactions to actions such that:

1. $sp(T_i) = o_k$ where $o_k \in T_i$; and
2. If $sp(T_i)$ occurs before $sp(T_j)$ in S, then there exists a serial schedule equivalent to S in which T_i precedes T_j. □

If the serialization point of T_i precedes T_j in S, then no dependencies of the form $T_j \rightarrow \cdots \rightarrow T_i$ are allowed in local serialization graph for S.

The class of sp-schedules is a proper subset of the class of strongly serializable schedules. To see that any sp-schedule is strongly serializable, consider an sp-schedule S. Say T_i precedes and does not overlap T_j. Note that $sp(T_i)$ must map to a T_i action, and hence $sp(T_i)$ must precede $sp(T_j)$. Thus, there is an equivalent serial schedule where T_i precedes T_j. To see that not all strongly serializable schedules are sp-schedules, consider the following example:

$$S:\ r_1(a)\, w_2(a)\, w_3(b)\, c_2\, c_3\, r_1(b)\, c_1$$

First, it is a serializable schedule, equivalent to T_3, T_1, T_2. Since every transaction is overlapped with the others, the schedule is strongly serializable. Unfortunately, there is no serialization point assignment for T_1. That is, $sp(T_1)$ should map to $r_1(a)$ to make sure that $sp(T_1)$ precedes $sp(T_2)$. At the same time, $sp(T_1)$ needs to map to $r_1(b)$ or c_1 so that $sp(T_3)$ precedes $sp(T_1)$. Thus, S is not an sp-schedule.

Since local sp-schedules are strongly serializable, it is possible to use the global concurrency control schemes outlined in the previous section. However, if each local DBMS notifies the GTM in advance what action will constitute the serialization point, then one could obtain global serializability more efficiently. For example, a timestamp scheduler might indicate that the first action submitted is the serialization point (i.e., when the transaction receives its timestamp). In this general model, each site could define a different action to be the serialization point. For example, one site could say first actions are serialization points (it runs a timestamp algorithm) and another site could say last actions are (it runs a two phase locking protocol).

The global concurrency control mechanisms can then be extended for this more general model [45]. As before, the key idea is that the serialization points for transactions that may lead to cycles are executed in the same order at all sites. To enforce this, an analogy can be drawn to centralized DBMS [45]: Each site s_k is viewed as a single data object, o_k. If a transaction issues actions at s_k, it is viewed as issuing actions on o_k. Two serialization point actions, $sp_i(o_k)$ and $sp_j(o_k)$, always conflict. Other actions do not conflict. If the GTM ensures that the schedule in the analogous model is serializable, then it ensures that the global schedule in the real system is globally serializable.

4.2.3 Strongly Recoverable DBMSs

If we restrict our notion of serialization points so that they must occur at the *end* of each transaction, i.e., at its commit action, then we can obtain a GTM that is more efficient that those for strongly serializable schedules. The following definition captures this notion. (Here and in the next section we consider general notion of schedule and their committed projections).

Definition 4.2 [6, 54]: We say that schedule S_k is *strongly recoverable* if, for all pairs of transactions T_i and T_j, if T_i is in direct conflict with T_j in S_k and T_j commits in S_k, then T_j does not execute its commit before T_i commits. (See definition of direct conflict in Section 2.) □

Every strongly recoverable schedule is also recoverable [7]. Indeed, let S_k be a strongly recoverable schedule. Let us assume that transaction T_j *reads-x-from* transaction T_i. Then, by definition of strong recoverability, if transaction T_j commits then it commits after transaction T_i commits in S_k and, therefore, satisfies a condition of recoverability. Since every strongly recoverable schedule is serializable and not every recoverable schedule is serializable, we obtain that a class of strongly recoverable schedules is a proper subclass of recoverable schedules.

Strongly recoverable schedules are a proper subset of sp-schedules. To see this, consider a strongly recoverable schedule S and let $sp(T_i)$ map to c_i. Consider two transactions T_i, T_j such that $sp(T_i)$ precedes $sp(T_j)$ in S, i.e., c_i precedes c_j. Suppose that there is *no* equivalent serial schedule where T_i precedes T_j. For this to be true, there must be dependency graph arcs $T_j \rightarrow \cdots \rightarrow T_i$. Because S is strongly recoverable, this means that c_j precedes the commit points of the intermediate transactions in the dependencies, which, in turn, precede c_i. Since we know that c_i precedes c_j, our supposition must be false. Thus, $sp(T_i)$ precedes $sp(T_j)$ in S implies there must be some serial schedule where T_i precedes T_j, so S is an sp-schedule. To see that strongly recoverable schedules are a proper subset of sp-schedules, consider the schedule:

$$S\colon\ r_1(x)\ w_2(x)\ c_2\ c_1.$$

Schedule S is an sp-schedule, but it is not strongly recoverable.

The various concurrency control mechanisms discussed in literature (i.e., 2PL, timestamp ordering [7], optimistic [38], etc.) can be easily modified to ensure that they generate strongly recoverable schedules [54, 12].

To see why the knowledge that local sites generate strongly recoverable schedules (as opposed to strongly serializable or sp-schedules) leads to higher concurrency of the global concurrency control mechanism, let us return to the strategy of executing global transactions serially. If local sites generate strongly serializable or sp-schedules, the GTM avoids cycles by making sure global transactions do not overlap (see Section 4.2.1). With strongly recoverable local schedules, however, it is sufficient to ensure that transactions do their commit processing serially (i.e., between the time a global transaction issues its first commit at a site and its last commit at another site, no other global transaction issues any commits). To explain why this works, let us return to a slightly modified version of Example 3.1:

$$S_1\colon\ r_1(a)\ c_1\ w_3(a)\ w_3(b)\ c_3\ r_2(b)\ c_2$$

$$S_2\text{: } w_4(c)\ r_1(c)\ r_2(d)\ c_2\ [\ w_4(d)\ c_1\ c_4\ ]$$

The actions in brackets again represent future actions. Here the actions of T_1 and T_2 are interleaved (i.e., T_2 reads at s_2 before T_1 commits). A site that generates strongly serializable or sp-schedules would permit the remaining actions to take place, leading to a non-serializable global schedule. However, a strongly recoverable site would not. Because of strong recoverability, sites will not allow paths of the form $T_2 \rightarrow \cdots \rightarrow T_1$. In this particular example, the path $T_2 \rightarrow T_4 \rightarrow T_1$ is not allowed because it would imply that c_2 precedes c_4 and c_4 precedes c_1 (impossible since c_1 precedes c_2). Thus, when T_1 attempts to commit in the example, it will either be aborted or to be left waiting until T_4 commits. Thus, the schedule above cannot occur.

A strongly recoverable local scheduler in essence is giving the GTM control of the serialization points. That is, when the GTM submits the commit for T_1, it knows that it will be serialized before any global transactions whose commits are submitted later.

This same idea can be applied to the other global concurrency control mechanisms of Section 4.2.1. For example, the site-graph algorithm [9] becomes the commit-graph algorithm [13]. In the commit-graph approach, the GTM maintains a commit-graph that is similar to the site-graph. Unlike the site-graph approach in which the edges corresponding to a transaction are inserted when the transaction starts execution, in the commit graph the edges corresponding to a transaction are inserted just before the commit process of the transaction is started. This permits all transactions to be executed concurrently, except during their commit phase. The commit phases of transaction that may be involved in a global cycle (as determined by the commit graph) are executed serially.

4.2.4 Rigorous DBMSs

Some local concurrency control mechanisms are even more restrictive than the ones we have reviewed so far, and can, in turn, lead to even more efficient global scheduling schemes. For example, consider an MDBS where all local sites use the strict two phase locking protocol [7], which is the most popular type of mechanism used in practice. In this environment, the following undesirable schedule of Example 3.1 cannot occur, even if *no* global concurrency control is present (the actions in brackets again represent future action):

$$S_1\text{: } r_1(a)\ [\ c_1\ w_3(a)\ w_3(b)\ c_3\ r_2(b)\ c_2\ ]$$
$$S_2\text{: } w_4(c)\ [\ r_1(c)\ c_1\ r_2(d)\ c_2\ w_4(d)\ c_4\ ]$$

At site s_2, T_4 keeps a lock on c until it commits. Hence, T_1 cannot read c, and is delayed. If T_1 runs after T_4 at S_2, then the undesirable dependency $T_4 \rightarrow T_1$ does not happen. Unfortunately, the use of the strict 2PL at each participating site does not automatically guarantee global serializability, as the next example illustrates.

Example 4.1: Consider a multidatabase system located at two sites: s_1 with data items a and b, and s_2 with data items c and d. Let T_1 and T_2 be two read-only global transactions, and let T_3 and T_4 be two local transactions. The schedules at sites s_1 and s_2 are, respectively:

$$S_1 :\ w_1(a)\, c_1\, r_3(a)\, w_3(b)\, c_3\, r_2(b)\, c_2$$
$$S_2 :\ w_2(x)\, c_2\, r_4(x)\, w_4(y)\, c_4\, r_1(y)\, c_1$$

At each site, the schedules can be produced by strict two phase locking (they are actually serial at each site). However, the dependencies $T_1 \rightarrow T_3 \rightarrow T_2$ and $T_2 \rightarrow T_4 \rightarrow T_1$ exist and the global schedule is not serializable.

This problem can be avoided if the GTM does not issue any commits for a transaction until all of its actions have been completed. In Example 4.1, the operation c_1 at s_1 would be delayed until

$r_1(y)$ at s_2 is acknowledged. In turn, T_3 would block until T_1 commits, and the above schedule could not take place. (It may lead to a global deadlock, though.) It can be shown that delaying the commits until a transaction completes all of its read/write actions is enough to guarantee global serializability. No additional synchronization between global transactions is required in this case.

The following definition captures what it is about strict 2PL that ensures global serializability.

Definition 4.3 [6]: We say that schedule S_k is *rigorous* if, for all pairs of transactions T_i and T_j, if T_i is in direct conflict with T_j in S_k and T_j commits in S_k, then T_j does not execute its conflicting operation before T_i commits. $\square$

In [6] it is shown that if local DBMS schedulers are rigorous and the GTM does not schedule the commits of a transaction until all previous operations of the same transaction have completed their execution, then the global schedule is serializable. As we have stated, the strict 2PL protocol generates rigorous schedules. Other protocols can be easily modified to generate rigorous schedules. For example, basic timestamp ordering can be made rigorous by blocking transactions that either try to read or write data which were previously written by uncommitted transaction or try to write data which were previously read by uncommitted transaction [6].

It is important to note that if local sites only provide a service request interface (Section 2), then the GTM *cannot* delay the commits as required in [6]. In Example 4.1, if the GTM sends a service request to s_1 on behalf of T_1, it is actually sending the entire subtransaction $w_1(a)\, c_1$, so c_1 cannot be delayed and this global schedule could occur. In this case, the GTM would have to use additional global transaction synchronization, as is done for strongly recoverable local schedules.

It is not hard to see that any rigorous schedule must be strongly recoverable. Say S is a rigorous schedule with a conflict between operations op_i and op_j where op_i occurs first in S. Because S is rigorous, c_i must precede op_j. And clearly op_j precedes c_j (see Section 2). Thus, c_i precedes c_j and S is strongly recoverable. Note that not all strongly recoverable schedules are rigorous. For example, the schedule S: $r_1(x)\ w_2(x)\ c_1\ c_2$ is strongly recoverable but not rigorous.

In summary, we have a hierarchy of local schedule classes, going from the most general to the most restrictive: serializable, strongly serializable, sp-schedules, strongly recoverable, and rigorous. Each class is a proper subset of the next most general one. We have also shown that as the local scheduler becomes more restrictive, the global GTM can be more permissive in coordinating global transactions operations. Indeed, we have seen that if every local DBMS generates a rigorous schedule, then the GTM does not perform any operation coordination. For the case of strongly recoverable local schedules, the GTM only needs to coordinate an execution of global transactions commit operations. If each local DBMS generates a sp-schedule, then the GTM needs to coordinate transaction serialization point operations for each local site. Finally, in the case of local strongly serializable schedules, the GTM should coordinate an execution of all operations of global transactions.

5 Alternative Consistency Notions

As we have seen, guaranteeing global serializability may result (in some environments) in poor performance due either to a low degree of concurrency or the large number of aborted transactions. Moreover, as we shall see later, when we discuss failures, it is very hard to obtain global serializability in some cases. Thus, several researchers have suggested notions of correctness that are weaker that global serializability. In this section we survey some of these notions, still assuming that neither failures nor unilateral aborts of global transactions can take place.

5.1 Local Serializability

Global serializability guarantees that all consistency constraints are satisfied. If global serializability is to be dropped, then it is important to guarantee consistency in some other way. Consistency is usually defined in terms of integrity constraints that must hold among various data items. Thus, it is important to study constraints and look for alternative ways of satisfying them.

In a MDBS, there are two types of constraints: *local* ones that involve data items located at a single site, and *global* ones that involve date items located at more than a single site. It can be argued that in some multidatabase applications there are *no* global constraints, since each site was developed independently, and sites may wish to remain independent [25]. For example, airlines run independent reservations systems with no global constraints among them. The systems do interact (e.g., at the reservation system of airline Y one can reserve a seat on airline X's flight), but each system only cares about the consistency of its own data.

Fortunately, local constraints are easy to maintain. Essentially, each site can run a local concurrency control mechanism that ensures that the local schedule is serializable. This, in turn, ensures (with one catch explained below) that all local constraints are satisfied, with no need for any global synchronization among sites. The resulting global schedule is not globally serializable, but is *locally serializable* as defined below [23, 39].

Definition 5.1: A global schedule S is locally serializable (LSR) if for every site s_i, the local schedule is serializable. □

Given an initial database state, an *execution* of a set of transactions $\mathbf{T}$ results in a final database state. An execution also produces a schedule which represents the sequence of read, write, abort, commit operations. For each transaction $T_i \in \mathbf{T}$, an execution also defines a database state read by each T_i.

Definition 5.2: We say that an execution is *strongly correct* if the final state produced is consistent and the state read by each transaction $T_i \in \mathbf{T}$ is consistent (i.e., any values read by T_i satisfy constraints that span them). □

To show that LSR schedules guarantee strongly correct executions, we need to rule out certain types of "unusual" transactions [48], as illustrated by the following example.

Example 5.1: Consider an MDBS where data item a is stored at site s_1 and data item b is stored at s_2. Suppose that we have two constraints: $a > 0$ and $b > 0$. Consider the following two transactions:

$$\begin{array}{ll} T_1: & a := -1 \\ & \textbf{if } (b > 0) \textbf{ then } a := 1 \end{array}$$

$$\begin{array}{ll} T_2: & b := -1 \\ & \textbf{if } (a > 0) \textbf{ then } b := 1 \end{array}$$

Note that both T_1 and T_2 are valid transactions: given any initial state that is consistent, they transform the database into a consistent state. Consider the following executions at the two sites. We use the notation $r_i(a, x)$ ($w_i(a, x)$) for a read (write) action of transaction T_i on item a, where x is the value read (written).

$$S_1: w_1(a, -1)\ r_2(a, -1)$$
$$S_2: w_2(b, -1)\ r_1(b, -1)$$

Each local schedule is serializable. Nevertheless, the final state a = -1, b = -1 is inconsistent. □

There are at least three ways to avoid the type of problems shown in Example 5.1 above:

1. *Force transactions to be Local Database Preserving.* If we look at T_1 (in example 5.1) from site s_1's point of view, the local actions of T_1 do not constitute a valid local transaction. That is, there is an initial state ($a > 0$, $b < 0$) that is *locally* consistent, and when T_1 runs, it transforms it into a locally inconsistent state (a = -1). Thus, from s_i's point of view, T_1 is breaking the rules: The correct operation of T_1 depends on the correctness of some data (b) over which s_1 has no control. The problem can be avoided if we require that any transaction that runs at site s_i preserves consistency regardless of the state of other sites. Such transactions are called Local Database Preserving (LDP) [47]. If transactions are LDP and all constraints are local, it is easy to show that LSR schedules guarantee strongly correct executions.

2. *Require local sites to run a two-phase locking (2PL) protocol.* In Example 5.1, if sites s_1 and s_2 were following the 2PL protocol, then transaction T_1 would not release the lock on a until after it has read the value of b (since it may require to write on data item a, if in case the value of $b > 0$) and T_2 would not release the lock on b until after it has read the value of a. As a result, T_1 will wait for T_1 to release the lock on b, and T_2 will wait for T_1 to release the lock on a thus resulting in a deadlock. Hence, the execution as in Example 5.1 will not be permitted.

3. *Force transactions to have a fixed structure.* A transaction that always has the same read/write pattern is termed a *fixed-structure* transaction [48]. That is, regardless of what it reads, it will read and write the same items, in the same order. Transaction T_1 in Example 5.1 is not fixed-structure: it may or may not write a. To make it a fixed-structure transaction, we could rewrite it as, say:

$$\begin{aligned} T_1: \quad & a := -1 \\ & \textbf{if } (b > 0) \textbf{ then } a := 1 \textbf{ else } a := -1 \end{aligned}$$

   In this case, the scenario shown in Example 5.1 could not arise. The schedule at site s_1 would be S_1: $w_1(a)\ r_2(a)\ w_1(a)$ and would be non-serializable. In general, it can be shown that if all transactions are fixed-structure (and constraints are local), then a LSR schedule guarantees that all executions are strongly correct.

The requirement that all transactions be LDP is not unreasonable. Local transactions are always LDP, so the requirement does not affect the autonomy of sites. Most practical global transactions will be LDP anyway; if not, they can be made LDP with little effort, provided that the local constraints are known. The constraint that the local schedulers follow the 2PL protocol is also reasonable since in any case most practical concurrency mechanisms follow the such a protocol. The requirement for all transactions to be fixed structure may, however, be less reasonable since it requires even local transactions to be fixed-structure which violates local autonomy. Note that transactions that only contains assignment and alternation statements can always be converted into fixed structure transaction. For example, as we illustrated earlier, the transactions in Example 5.1 could be made fixed-structure. However, if transactions contains loops, there may not be an easy way to make them fixed structure.

As a historical note, a fourth strategy has been suggested for making LSR schedules preserve constraints. A transaction T_i is NVD if it has *no value dependencies* [17]; that is, if its actions at a site never depend in any way on the values read at another site. Both T_1 and T_2 in Example 5.1 have value dependencies. If a transaction is NVD, then it is clearly LDP. The converse is not true (example: a transaction that writes into item b a value read elsewhere, but b is not involved in any constraints). Making transactions NVD ensures executions are strongly correct (because transactions are LDP), but as noted in [17] this is more restrictive than necessary.

5.2 Two Level Serializability

The notion of local serializability can be extended as follows. There are two types of data at each site: *local data* and *global data*. Three types of constraints are allowed:

1. *Local.* Constraints involving local items; each local constraint can involve only items at a single site.

2. *Global.* Global constraints may span more than one site, but can only involve global items.

3. *Global/Local.* These constraints can only span a single site, but may involve both local and global items.

The main restriction on transactions in this model is that local transactions may not modify global data. (Global data is usually involved in inter-site constraints; a local transaction would be unable to maintain these since it can only run at a single site.) For now, no other access restrictions are made: local transactions can read both local and global data, and global transactions can read and write any data.

This extension is applicable to an MDBS environment that started as a collection of independent databases. These original databases constitute the local data, and original transactions only access local data. A new data layer was then added, the global data. It is stored in the same DBMSs at each site, except that it is managed by newer transactions that are run through the GTM (global transaction manager). Since the new transactions run under the control of the GTM, it is now feasible to enforce global constraints that span the new data. The new transactions are allowed to read and write the original local data. Finally, for efficiency, we might want to add a third class of transactions, new local ones. These are run by the local DBMS but are allowed to read the new global data.

It is important to note that local and global/local constraints should not involve remote data, even indirectly. For instance, say a_1 is a local item, and b_1 and b_2 are global ones. Item b_2 is at site s_2; a_1 and b_1 are at s_1. Also assume we have constraints $a_1 = b_1$ and $b_1 = b_2$. The global/local constraint $a_1 = b_1$ is not allowed because it induces constraint $a_1 = b_2$ which relates a local item to a remote item. Not all global/local constraints cause this problem. For example, consider the constraints a_1 ¿ b_1 and b_2 ¿ b_1.

Since the GTM controls global transactions, it can ensure that the global schedule, as far as access by these transactions is concerned, is serializable. Similarly, the local concurrency control mechanisms will ensure that the local schedules are serializable. This motivates the notion of two-level serializability.

Definition 5.3 [48]: A global schedule S is *two level serializable (2LSR)* if it is LSR and its projection to a set of global transactions is serializable.

Globally serializable schedules are always 2LSR, but the converse is not true. This is illustrated by the following example, which also shows that 2LSR schedules may violate constraints if they contain "unusual" transactions:

Example 5.2 [48]: Consider an MDBS where there is a single local item a at site s_1 and three global items, b and c at s_1 and d at s_2. There is one global/local and one global constraint:

$$a > 0 \rightarrow b > 0$$
$$d > 0 \rightarrow (b > 0 \text{ or } c > 0)$$

Consider the following two global and one local transactions:

$$
\begin{aligned}
T_1: \quad & \textbf{if } (a \ <= \ 0) \textbf{ then } c := 1 \textbf{ else } c := -1 \\
& d := 1 \\[1em]
T_2: \quad & \textbf{if } (a \ <= \ 0) \textbf{ then } b := -1 \textbf{ else } b := 1 \\
& d := -1 \\[1em]
L_3: \quad & a := -1
\end{aligned}
$$

Starting from a state where all items have a value of "1", consider the following executions:

$$
\begin{aligned}
S_1: \quad & r_1(a,\ 1)\ w_1(c,\ -1)\ w_3(a,\ -1)\ r_2(a,\ -1)\ w_2(b,\ -1) \\
S_2: \quad & w_2(d,\ -1)\ w_1(d,\ 1)
\end{aligned}
$$

The resulting state is a = -1, b = -1, c = -1, d = 1, which is inconsistent. Note that the global schedule is 2LSR but is not globally serializable (at s_1 we have dependency $T_1 \rightarrow L_3 \rightarrow T_2$, and at s_2 we have $T_2 \rightarrow T_1$). □

The problem in this case is with the transactions. From the point of view of each site, transactions are LDP, so all local and global/local constraints are satisfied. However, the global constraint is being violated because from the point of view of the global data, T_1 is not a proper transaction. Transaction T_1 only produces a consistent state when an condition external to the global data ($a > 0 \rightarrow b > 0$) is satisfied. (In particular, say we start with the state $a = 1$, $b = c = d =$ -1. As far as the global constraints are concerned, it is consistent. Yet T_1 will transform it into an inconsistent global state.) If T_1 were rewritten more sensibly as T_1: $c := 1$; $d := 1$, then consistency would be preserved (T_1 now enforces global constraints regardless of the state of other constraints).

We define a transaction to be Global Database Preserving (GDP) if it preserves global constraints regardless of the state of local data items. This notion is analogous to LDP. If we think of the global items as constituting a single database at an imaginary site α then saying a transaction is GDP is equivalent to saying it is LDP at α. It is not hard to see that if all transactions are LDP and GDP, then a 2LSR schedule guarantees all executions are strongly correct.

If we make certain additional assumptions about the access patterns of transactions, then it is possible to relax the LDP, GDP requirement [48]:

1. If global transactions are not allowed to access local data, then we can drop the GDP requirement. (Actually, if global transactions cannot read local data, then they are necessarily GDP. So the requirement is not dropped; it is replaced by a more restrictive one). If we further assume that local transactions cannot read global data, and that the global transactions do not write local data, then the local and global data is totally decoupled; 2LSR schedules will always be serializable, without any requirements on the transactions.

2. If there are no global/local constraints, then the GDP requirement can be dropped. The proof of this is lengthy [47], but the intuition is as follows. Since transactions are LDP, if the local schedules at each site are serializable, then the local constraints will be preserved (regardless of whether transactions are GDP or not). Also, the state of local data seen by global transactions will be consistent. Furthermore, since the projection of the schedule onto global transactions is serializable, the state of global data items in which a transaction executes is consistent. Thus, since there are no global/local constraints, the state in which a global transaction executes along with the values of local data item it sees is consistent.

(Contrast this to the execution in Example 5.2. In the example, T_2 is serialized before T_1 in the projection of the schedule onto global transactions. Thus, the state of global data items in which the transaction T_1 executes is b = -1, d = -1, c = 1. Since T_1 reads the value of a to be 1, it is inconsistent with the value of b = -1.)

Once we assume that there are no global/local constraints, the remaining LDP constraint can be relaxed as follows:

1. If global transactions cannot access local data, it can be shown that the LDP requirement can be replaced by one forcing all transactions to have fixed structure.

2. If local transactions cannot read global data, and global transactions cannot write local data, then the LDP requirement can be dropped entirely. In this case, activity on the local data is completely decoupled from the activity on the global data. Hence, the fact that local schedules are consistent is sufficient to guarantee that local constraints are always true.

3. If local schedulers are 2PL and *global* transactions are fixed structure, then there is a GTM locking strategy (called two-level two-phase locking) that can ensure constraints are satisfied [47]. Note that since only global transactions are required to be fixed structure, local autonomy is not violated.

As a historical note, we point out that a precursor to the notion of 2LSR schedules was the notion of quasi-serializablity [17, 16]. A global schedule is said to be *quasi serial* if and only if is LSR and there is a total order of global transactions such that for any two global transactions T_i and T_j if T_i precedes T_j in the total order, then all T_i operations precede all T_j operations in all local schedules in which both appear. A global schedule is *quasi serializable (QSR)* if it is equivalent to a quasi serial schedule. The QSR class is a proper subset of 2LSR. (Example 5.2 shows a 2LSR schedule that is not QSR.) It appears that there are no advantages to restricting schedules from 2LSR to QSR [48], so QSR schedules seem to be only of historical interest.

5.3 Other Constraint Based Criteria

With 2LSR schedules, the GTM ensures that any global constraint is satisfied by executing the global transactions is such a way that the resulting schedule (of actions of global transactions) is serializable. A different idea for enforcing global constraints is presented in [5]. The claim is that global constraints tend to be very simple in practice and that the GTM can enforce them directly, without concerning itself with serializability. A second claim is that global constraints tend to be "approximate," giving the GTM even more flexibility in enforcing them.

To illustrate, consider a copy constraint between item g_1 at site s_1 and g_2 at s_2. Many applications, especially if they run on independent sites, can tolerate some divergence, e.g., the copy constraint may be $|g_1 - g_2| \leq \epsilon$, where ϵ is some application dependent value. In this case, not every update to g_1 needs to be reproduced at s_2 and vice versa. The server at s_1 (see Figure 1) can keep track of a window of allowable values for g_1, and while g_1 remains in this window, copies of the new values are not propagated to s_2. The advantages of this added flexibility will be more apparent when failures are considered in Section 6.

In summary, in [5] global schedules are assumed to be LSR, so all local (and global/local) are satisfied. Even though global schedules are not 2LSR, global constraints are enforced "manually" by the GTM and its servers. It is assumed, of course, that applications declare a priori their global constraints. If the constraints fall outside of the repertoire of the GTM, then it reverts to enforcing 2LSR schedules.

5.4 Limitations of Constraint Based Approaches

In the previous subsections we have described alternative correctness notions based upon preservation of the database consistency constraints. Each of the correctness notions (LSR,or 2LSRR) can be shown to preserve strong correctness of schedules under appropriate restrictions. We have, however, avoided the question of whether the preservation of strong correctness is a sufficient consistency guarantee for transactions. The answer to this is application dependent. While a strongly correct schedule preserves all the database consistency constraints, it may, however, not be sufficient for preventing all undesirable executions in certain applications, as illustrated below.

Example 5.3 [49]: Consider a banking database located at two sites: s_1 with account a, and s_2 with account b. Suppose that we have constraints specifying that no account have a negative balance. Consider transaction T_1 that transfers 500 dollars from account a to account b, and an audit transaction T_2 that reads the balance of both a and b. Transaction T_1 consists of two subtransactions, a debit subtransaction and a credit subtransaction,

debit: **if** $a > 500$ **then** $a := a - 500$
else abort

credit: $b := b + 500$

Consider the following schedule:

$$S_1:\ r_1(a)\ w_1(a)\ r_2(a)$$
$$S_2:\ r_2(b)\ r_1(b)\ w_1(b)$$

Transactions see non-negative balances, and the final state is also consistent, so the schedule is strongly correct. However, in this schedule, the audit transaction sees 500 dollars less than the actual sum total of accounts a and b. This may be considered an "anomaly" and thus it may not be sufficient for schedules to be strongly correct. □

Note that in Example 5.3, the execution is neither 2LSR nor QSR (though it is strongly correct). Examples in which undesirable executions occur (even though the execution is 2LSR and/or QSR) can be similarly constructed.

In Example 5.3, we could say that there is a second type of correctness criteria, in addition to strong correctness. In this case we do not want the transfer transaction to be involved in any serialization cycle. One "artificial" way of dealing with this problem is to declare another data item *total* and define an integrity constraint $total = a + b$. If this constraint were defined, then the schedule of Example 5.3 would not be strongly correct and would be avoided.

However, one could argue that defining additional constraints is not desirable. First, there may be no real integrity constraint between accounts a and b; that is, if any other transaction sees the value of a and b not equal to *total*, that may be quite acceptable. It is only audit transactions that are special. If we declare the constraint, we will disallow many executions, not just those involving audits. Second, if we were to declare such constraints, we would need to declare data integrity constraints between every two (or in general n) accounts. This will result lost concurrency, and, in general, will reduce strong correctness to serializability; that is, the only strongly correct schedules will be serializable ones.

Thus, in addition or instead of ensuring strong correctness, it may be useful to develop mechanisms that restrict schedules in some way, without requiring serializability. This is discussed in the next section.

5.5 Non-Constraint Based Criteria

Pu and Leff [52] introduced a notion of *Epsilon-Serializability* as an alternative correctness notion. Transactions are divided into read-only and update transactions. The execution of the update transactions is assumed to be serializable, so that database consistency is preserved. However, the full schedule of all transactions is allowed to be non-serializable, as long as "the number of non-serializable conflicts is limited." To illustrate how conflicts are counted, consider the schedule:

$$w_2(x)\ r_3(x)\ r_3(y)\ w_1(y)\ r_1(z)\ w_2(z)$$

The schedule of update transactions is equivalent to the serial schedule $< T_1, T_2 >$. However, T_3 breaks this order by reading from T_2 but not from T_1. (That is, T_3 is involved in the cycle $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_2$.) This is counted as T_2 exporting one conflict and T_3 importing one conflict. (T_1 does not export any conflicts.) The limits on conflicts are given by *import and export limits*: each read-only query has an import limit specifying how many conflicts it can be involved in; each update transaction is given an export limit giving the maximum number of conflicts. This idea is formally captured in the definition below.

Definition 5.4: A schedule S is ϵ-serial is its projection on the update transactions is serial, and the number of conflicts imported by each read-only transaction does not exceed its import limit, and the number of conflicts exported by an update transaction does not exceed its export limit. A schedule is ϵ-serializable if it is equivalent to a ϵ-serial schedule. □

Note that if the limits of all transactions are set to zero, then ϵ-serializable schedules are serializable. Several methods to control consistency divergence are proposed in [68]. One of these methods uses an extension of two phase locking. Read-only transactions are allowed to read data locked by updates, but each such access counts as a conflict. If the limits are reached, then such accesses are disallowed. Under these conditions the protocol ensures that schedules are ϵ-serializable [68].

One weakness of the original ϵ-serializability approach is that it does not tell us how corrupted read data may be. For instance, suppose that we have the constraint $a + b = c$ on three bank accounts. A single conflict violation may cause a transaction to read values such that $a + b$ is a trillion dollars larger than c. The approach was extended in [68] so that conflicts are measured in a way that is more meaningful to the application.

A different notion of correctness is used in [26]. Here transactions are grouped into disjoint types. An application administrator then determines that transactions of certain types can be interleaved arbitrarily without causing constraints to be violated. For example, in a bank it may be safe for deposit transaction to interleave with other deposits and with transfer transactions. The concurrency control mechanism proposed in [26] uses local locks to ensure LSR schedules, and global locks to avoid undesirable interleavings.

The concept of compatibility is refined in [43] and several levels of compatibility among transactions are defined. These levels are structured hierarchically so that interleavings at higher levels include those at lower levels. Furthermore, [43] introduces the concept of breakpoints within transactions which represent points at which other transactions can interleave. This is an alternative to the use of compatibility sets. A similar scheme that uses breakpoints to indicate the interleaving points, but does not require that the interleavings are hierarchical is presented in [19].

In [49], the approach taken is to classify the global transactions into two classes: *RS-transactions* and *non RS-transactions*. The GTM protocol proposed in [49] ensures that no cycle in the serialization graph of a global schedule contains any RS-transaction (in addition to ensuring that schedules are strongly correct). Returning to Example 5.3, the audit transaction T_2 is an RS-transaction, whereas the transfer transaction T_1 is a non RS-transactions. Thus, even though the schedule S_1 is strongly correct, it is not permitted since the serialization graph of S_1 contains a cycle involving transaction T_2 which is an RS-transaction.

6 Atomicity and Durability

In this section, we discuss how transaction atomicity and database consistency can be preserved in presence of global transactions aborts and failures. In a multidatabase system, as in a homogeneous systems, failures may range from transaction aborts, systems failures, failure of the GTM, to link and communication failures. In addition, in a multidatabase system, a global transaction at a local site can be aborted by a local DBMS as a result of normal DBMS operations (such as aborts caused by a local deadlock detection procedure) and the same transaction can be committed at some other local sites. In this paper we consider such situations as global transaction failures. Multidatabase recovery procedures should ensure that the GTM can recover from both these unilateral aborts and from failures. Since the recovery procedures at each local DBMS ensure atomicity and durability of local transactions and global subtransactions, the task of ensuring atomicity and durability of transactions in a distributed system reduces to ensuring that each global transaction either commits at all the sites, or it aborts at all the sites.

The key factor that effects the design of the GTM recovery procedures is the interface provided by the local DBMSs. As we mentioned in Section 1, there is an ongoing debate among researchers whether or not local DBMSs will provide a prepare-to-commit operation for the transactions. If each local DBMS provides such an operation, then the task of ensuring atomicity is relatively simple since an atomic commit protocol (e.g., 2PC protocol) can be used. This is discussed in Section 6.1. On the other hand, if local DBMSs' do not support a prepare-to-commit operation, then it is possible that a global transaction commits at some sites and aborts at others. In this case, three different mechanisms for ensuring global transaction atomicity have been studied in the literature.

1. **Redo.** The writes of the failed subtransaction are installed by executing a *redo transaction* consisting of all the write operations executed by the subtransaction.

2. **Retry.** The entire aborted subtransaction, and not only its write operations, is run again.

3. **Compensate.** At each site where a subtransaction of a global transaction did commit, a compensating subtransaction is run to semantically undo the effects of the committed subtransaction.

We discuss these approaches in Sections 6.2 through 6.4. While redo and retry techniques ensure the standard atomicity of transactions, in the case of compensation a weaker notion of atomicity is used, since it is possible that the effects of the aborted global transaction are externalized to other transactions. This impacts the preservation of consistency in the systems. We will also discuss this issue in Section 6.4. Finally, each of the above techniques are complementary; that is, it is possible to combine them into a single uniform solution. We discuss how this can be done in Section 6.5.

6.1 Two Phase Commit

If the local DBMSs support a prepare-to-commit operation, then transaction atomicity and database consistency in a failure prone environment can be ensured by augmenting the various concurrency control mechanisms of Section 4 (or Section 5), with the use of the *two phase commit* (2PC) protocol [7] (or one of its variations). The 2PC protocol works as follows. At the termination of the execution of the transaction's operations, the GTM submits a prepare-to-commit operation to each site the transaction executed. On receipt of the prepare-to-commit operation, a site votes to commit or to abort the transaction. If the site votes to commit the transaction, it enters a *prepared state* for the transaction. On entering the prepared state, the site cedes its right to unilaterally abort the transaction to the GTM. The GTM, on receipt of the votes from each site the transaction executed at, depending upon the votes, either decides to commit or to abort the transaction. Each site in the prepared state complies with the decision of the GTM.

For the 2PC scheme to work, we require that before a site enters a prepared state, it must be in a position to commit or abort the transaction as instructed by the GTM (even in the presence of failures). Since it is possible that a system crash may occur while the site is in a prepared state for a transaction, the site must store the updates made by the transaction onto stable storage before entering the prepared state. Further, since other transactions may abort while the site is in a prepared state, it must also ensure that such aborts do not jeopardize its ability to comply with the GTM's decision.

To achieve this, before entering a prepared state for transaction T_i, the site s_k must ensure that each transaction T_j from which T_i has read some data item at s_k, is committed. Else, it is possible that the GTM decides to commit T_i, but since T_i read a data item at site s_k written by T_j that is aborted, T_i can no longer be committed by the local DBMS at s_k. If the local DBMS produces serializable and recoverable schedules [7], then the above property is ensured. Note, that a class of serializable and recoverable schedules is a proper superclass of the class of strongly recoverable schedules (Section 4.2.3). If, in addition, the local DBMS produces cascadeless schedules [7], then the GTM can submit a transaction commit as soon as the transaction has completed its read/write operations at each local site. In the latter case, the above property will be trivially ensured.

Another practical issue that must be addressed is that of heterogeneous commit protocols. To illustrate the problem, suppose that the interface of one local DBMS supports operations that are compatible for the execution of a two phase commit protocol, whereas another local DBMS's interface supports operations compatible for the execution of a three phase commit (3PC) [7]. The semantics of these operations and the actions taken by the local DBMS on their execution may be completely different. Thus, combining such local DBMSs to support a global commit protocol is a non-trivial task. In addition to differences in the operations supported by the local DBMSs for committing global transactions, there may be many other implementation level differences among sites, regarding issues such as error handling and who controls the global commit. If there were a single standard 2PC protocol, these problems would be avoided, but it is unlikely that this will occur. Already there are several competing "standards" (e.g., LU6.2 [15], OSI TP [62]). Thus, the problem of coordinating heterogeneous commit protocols will persist. Some initial work on such coordination is reported in [36].

As we argued in Section 3, there may be cases where the prepare-to-commit operation is not provided by all sites. This may be due to the following:

1. Sites only offer a Service Request interface, giving remote clients a set of services but not control over service commitment;

2. Sites wish to retain their execution or communication autonomy; or

3. Performance of 2PC in a distributed system may be inadequate. In particular, sites may have to remain in the prepared state for too long, blocking local resources; transaction response time and throughput may suffer because of this [5].

In the rest of this section we consider systems where no global atomic commit protocol (2PC) is being used.

6.2 Redo Approach

Consider the situation in which the local DBMSs do not support a prepare-to-commit operation. In this case, to ensure global transaction atomicity, the GTM may still use the 2PC protocol, where the servers (see model definition in Section 2) rather than the local DBMSs act as the participants. Since local DBMS do not support a prepare-to-commit state, the global transaction may be aborted at the local DBMS at any time, even after the server has voted to commit the transaction. If a global subtransaction is aborted by the local DBMS after the GTM has decided to commit the transaction, the server at the site at which the subtransaction aborted submits a *redo transaction* consisting of all the writes performed by the subtransaction to the local DBMS for execution. Note that to be able to construct such a redo transaction, the server must maintain a *server log* in which it logs the updates of the global subtransactions. In case of failure of the redo transaction, it is repeatedly resubmitted by the server until it commits. Since the redo transaction consists of only the write operations, it cannot logically fail.

If the above global commit protocol is used to ensure the atomicity of global transactions, then it must be the case that before the server sends to the GTM its vote to commit transaction T_i, each transaction from which T_i read some data item should have previously committed. If this were not the case, then it is possible that the GTM decides to commit a global transaction that reads data items written by an aborted transaction. If T_i only reads data item written by global transaction, then the server can ensure this property by delaying its vote until all such transactions from which T_i read some data items are committed. However, since T_i could have read a data item written by local transactions, and since the servers have no control over the execution of local transactions, this property, in general, can only be ensured if the schedules generated by the local DBMS are cascadeless [7] (note that in the case of homogeneous distributed database system, it is sufficient to ensure that each local DBMS is just recoverable!). Thus, for the above commit protocol to be applicable, the schedules produced by the local DBMS are required to be cascadeless.

Another problem with the redo approach is that since the local DBMS considers the redo transaction as a different transaction than the global transaction, the resulting local schedule may be non-serializable from the MDBS viewpoint. This was illustrated in Example 3.2 in which redo transaction T_3 is executed to redo the write operations performed by the globally committed but locally aborted transaction T_1. In that example, since the local DBMS considered T_3 as a different transaction than T_1, the resulting local schedule was not serializable from the GTM view point. Note that each of our correctness criteria discussed in Section 4 and 5 (that is, global serializability, LSR, or 2LSR) requires that the schedules at the local DBMSs be serializable from the MDBS point of view. We refer to the local schedule as being *m-serializable* [46], if it is serializable from the MDBS point of view. M-serializability can be defined as follows.

Definition 6.1: Let S_j be a local schedule consisting of local transactions, global subtransactions and redo transactions. Let $m(S_j)$ be a projection of S_j over committed transactions and also over the read operations performed by the global transactions that are aborted by the local DBMS but are committed by the GTM. Let T_i be such a transaction. In $m(S_j)$, reads performed by T_i and the write operations belonging to the redo transaction that executed to redo the updates of T_i is considered as a single transaction. We say that S_j is m-serializable, if and only if $m(S_j)$ is serializable. □

For example, schedule S_1 of Example 3.2 is not m-serializable. In order to ensure database consistency, the redo technique must be combined with techniques of ensuring m-serializability of the local schedules. To do so, let us consider the global/local data model described in Section 5.2. Let us assume that the following condition holds:

Global transactions that read local data items at site s_k do not write any local data items at s_k that can be accessed by local transactions at s_k.

If the global transactions are restricted as above, then the non m-serializable schedule S_1 of Example 3.2 will not occur. To see this, note that in Example 3.2, since the local transaction T_2 wrote data item a, it must be the case that a is a local data item. Since the global transaction both read and wrote data item a, it violated the above restriction, thus resulting in a non m-serializable execution. Unfortunately, even if the global transactions are restricted as above, m-serializability may not be ensured. In [46], it is shown that to ensure m-serializability, the local schedules besides being cascadeless must further be strongly recoverable, and that the GTM must ensure that the projection of the schedule over the global transactions' operations is rigorous.

In [46], it is further shown that the requirement of local schedules to be strongly recoverable and cascadeless, and of the projection of the schedule to global transactions to be rigorous can be relaxed, if we further restrict the data items accessed by global transactions as follows:

Global transactions that read local data items at site s_k do not update any data item at s_k.

We have discussed so far how, in the presence of failures, to ensure m-serializability if we were to use the redo technique to ensure the atomicity of global transactions. To ensure consistency, however, there is a need to further guarantee that one of the correctness criteria discussed in Section 4 and 5 is met. We would expect that if we were to augment our mechanism for ensuring the correctness criteria (that is, global serializability, LSR, 2LSR, etc.) in the absence of failures with techniques for ensuring m-serializability of local schedules, then it would suffice to achieve a solution for ensuring that global schedules satisfy the correctness criterion even in presence of failures.

This is in fact true if we were to choose either of LSR or 2LSR as our correctness criterion. Note that a global schedule is LSR if each of the local schedules are serializable. Thus, ensuring m-serializability of the local schedules suffices to ensure LSR of the global schedule in the face of failures. If we were to ensure the 2LSR correctness criterion of global schedules, then besides ensuring m-serializability, we must further ensure that the projection of the global schedule onto operations belonging to global transactions (which we refer to as GS) is also serializable. Note that to ensure m-serializability itself, the GTM needs to ensure rigorousness of GS (in the case of the first more weaker restriction on the interactions between global and local transactions). Since every rigorous schedule is also serializable, 2LSR is trivially ensured. On the other hand, if we were to adopt the more restrictive second restriction on global transactions, then to ensure that the global schedules are 2LSR, besides ensuring m-serializability, we will further need to ensure that the schedule GS is also serializable. We can do so by ensuring rigorousness of GS. However, depending upon the concurrency control protocol used to ensure serializability of GS, it may be possible to ensure both m-serializability of the local schedules and serializability of GS without ensuring rigorousness of GS. This problem still remains open.

If we were to choose global serializability as our correctness criterion, then simply augmenting the GTM concurrency control protocols developed in Section 4 with techniques of ensuring m-serializability may *not* suffice to ensure global serializability in the presence of failures. To see this consider the case when each local DBMS produces rigorous schedules. In that case, as discussed in Section 4.2.4, if the GTM does not issue any commits for a transaction until all of its actions have been completed, then global serializability is ensured in the absence of failures. Further, to ensure m-serializability we only need to ensure that the schedule GS is rigorous. This, however, may not ensure global serializability in the presence of failures as it is demonstrated by the following example.

Example 6.1: Consider a multidatabase system located at two sites: s_1 with global data items x and y, and s_2 with global data items u and v. Let T_1 and T_2 be global transactions and T_3 and T_4 be local transactions that execute at sites s_1 and s_2.

$$\begin{array}{ll} T_1: & w_1(x)\, w_1(u) \\ T_2: & w_2(y)\, w_2(v) \\ T_3: & r_3(x)\, r_3(y) \\ T_4: & r_4(u)\, r_4(v) \end{array}$$

Suppose that the GTM decides to commit both T_1 and T_2, but the local DBMS at s_1 aborts T_1. Thus, the following redo transaction T_5 is executed to redo the updates of T_1:

$$T_5\text{: } w_5(x)$$

The above execution results in the following local schedules S_1 and S_2 at sites s_1 and s_2 respectively.

$$\begin{array}{ll} S_1: & w_1(x)\, a_1\, w_2(y)\, c_2\, r_3(x)\, r_3(y)\, c_3\, w_5(x)\, c_5 \\ S_2: & w_1(u)\, c_1\, r_4(u)\, r_4(v)\, c_4\, w_2(v)\, c_2 \end{array}$$

Note that the above schedule is not globally serializable even though local schedules are m-serializable. □

The problem with the above situation is that it is possible that two global transactions that were not conflicting may indirectly conflict through local transactions due to the presence of failures. So one option of ensuring global serializability is to disallow reads of global data items by local transactions and writes of local data items by global transactions (while still disallowing reads of local data items by global transactions); as mentioned in section 5.2, this guarantees globally serializable schedules.

Another option of ensuring global serializability is to use some mechanism for preventing cycles in the global serialization graph through indirect conflicts between global transactions. Note that as discussed in Section 4, executing global transactions serially, or using one of the site locking, altruistic locking or the commit graph approach can be used for this purpose. The scheme developed in [10, 11] uses the commit graph approach to prevent cycles through indirect conflicts. Further, it is assumed there that local DBMSs follow the strict 2PL protocol (and thus produce rigorous schedules) and rigorousness of GS is ensured (by maintaining global locks and following the strict 2PL locking scheme on global locks) to ensure m-serializability of the local schedules.

Wolski and Veilainen also propose a related solution [67]. It is called the 2PC agent method, and it assumes that the participating local DBMSs produce only strict schedules. Their method, however, requires that global transactions cannot have indirect conflicts at local sites, which is equivalent to saying local transactions do not read global data. In subsequent work [63], the authors extend their method to multidatabase systems with only rigorous local DBMSs.

Note that if the local DBMSs do not ensure rigorousness of schedules, then since global transactions may conflict indirectly through local transactions, the mechanism for ensuring global serializability in the absence of failures will itself either prevent cycles through indirect conflicts or will detect such cycles. In case the scheme prevents such cycles (as is the case for the site graph approach, altruistic locking and the site lock approach) we conjecture that making the scheme failure-resilient will be relatively simple. The reason for our conjecture is that the scheme that prevents cycles through indirect conflicts in the absence of failures can be easily modified to also prevent cycles through indirect conflicts that may be caused due to the presence of failures. Thus, to make the scheme failure-resilient we only need to ensure that the local schedules are m-serializable. On the other hand, making schemes that detect such cycles failure-resilient may turn out to be more difficult since simply augmenting the scheme with mechanisms to ensure m-serializability may not guarantee global serializability.

The results we discussed so far indicate some weaknesses of the redo approach, namely, some restrictions needs to be imposed on data access by local and global transactions, which may not be suitable for certain applications. It appears that these restrictions are unavoidable, if execution autonomy of the local DBMSs is to be preserved. One way, however, of removing these restrictions is to exploit the semantics of the transactions for the purpose of recovery. We discuss this issue in the following subsections.

6.3 Retry Approach

Consider global transaction T_i that executes at two sites s_1 and s_2. On the completion of the transaction's operations, the GTM sends a "prepare" message to each server at sites where the transaction was executing. The server, on receipt of the prepare message sends the commit operation for T_i to the local DBMS. It is possible that T_i commits at s_1 and aborts at s_2.

Thus, the atomicity of T_i has been violated and because global transactions may read and write local data, the redo approach of Section 6.2 cannot be used. There are two options in this case: *retry* and *compensate*. In this section we consider the *retry* option and in the next section we consider the *compensate* one.

To ensure atomicity of T_i, one option that the GTM has is to resubmit the failed subtransaction, T_i^2, at s_2 as a new subtransaction T_i^3. This is not a matter of simply reproducing the writes of T_i^2; T_i^3 needs to be *run*, reading and writing possibly different values. This can only be done if the GTM saved the execution state of T_i (e.g., local variables in the program that executes T_i) that were used by T_i^2. It is also important that the original values read by T_i^2 were not communicated to other T_i subtransaction, since those reads are now invalid. In other words, there are no data dependencies between T_i^2 and any other subtransaction of T_i. Techniques such as [40, 34] can be used for checkpointing transaction programs and tracking data dependencies among subtransactions.

Further, it must be the case that subtransaction T_i^2 is *retriable* [50]; that is, if T_i^2 is retried a sufficient number of times (from any database state) it will eventually commit. This is important since before the subtransaction is retried the state of the local DBMS may be changed due to the execution of other local transactions. This should not result in the situation that the subtransaction cannot be committed. It must be noted that not every transaction satisfies this property. Consider, for example, a subtransaction that is to debit money from a bank account. Such a transaction, if retried, depending upon the balance in the account, may not successfully complete. On the other hand, if a subtransaction is to credit money into a bank account, then we can safely assume that if it is retried a sufficient number of times it will eventually successfully complete.

The technique discussed above describes how the retry approach can be used to ensure the atomicity of global transactions. In presence of multiple global transactions, in order to ensure database consistency, we will need to augment the retry technique with concurrency control mechanisms as discussed in Section 4 and 5. If the correctness criteria being ensured is LSR, then since after the transaction being retried has successfully committed, each local schedule is serializable, no concurrency mechanism is required by the GTM. On the other hand if we were to ensure 2LSR, then we will need to use one of the protocols to ensure that the projection of the schedules over the operations belonging to global transactions is serializable. Note that the GTM must consider the transaction executed to retry the aborted subtransactions as one of the subtransactions of the original global transactions. We conjecture that ensuring 2LSR is going to be relatively simple. If we were to use the retry technique and ensure global serializability, then as in the case of redo approach, due to the presence of failures, there may be indirect conflicts between global transactions through the local transactions. Thus, we will need to use one of the techniques discussed in Section 4 (that is, the site graph approach, site locking technique etc.) to prevent cycles from forming through such conflicts.

Thus, in general, the retry technique can be used for ensuring atomicity of transactions under the restrictions that subtransactions do not have data dependencies and that each subtransaction is retriable. The above scheme for ensuring atomicity was first mentioned in [44]. It is clear that due to restrictions that need to be imposed upon the transactions, the retry approach by itself is of limited applicability. However, as we will see later since it is possible to use each of the approaches discussed in conjunction, it may provide us with a powerful model. In any case, whenever transactions satisfy the required restrictions, the system should be capable off exploiting the retry approach.

6.4 Compensate Approach

Consider again the situation in the previous subsection in which a transaction T_i is committed at site s_1 and aborted at s_2. In contrast to the retry approach, another alternative is to *compensate* for the committed subtransaction T_i^2. This may be done by executing a *compensating transaction* CT_1^1 at site s_1, that undoes, from a semantic point of view, what T_1^1 did. For instance, if T_1^1 had reserved a seat for a given flight, CT_1^1 would cancel that reservation. Since the effects of the transaction have been externalized to other local transactions, the resulting state may not be the same as if T_1^1 had never executed but will be semantically equivalent to it.

To see this, consider for example, that transaction T_1^1 had reserved the last available seat for the flight. In that case, another transaction, say T_2, that tries to reserve a seat will be refused a reservation since the flight is already full. Had T_1^1 not executed T_2 would have been able to procure the reservation. Thus, the state that results after the execution of CT_1^1 differs from the state that would have resulted had T_1^1 not executed at all. This, as in the current flight reservation systems, is nevertheless quite acceptable.

We stress that compensating transaction for a committed global subtransaction is by itself a regular transaction and, thus, it must preserve database consistency. For this purpose, it may not only consist of an inverse function of the original subtransaction but may also consist of certain other actions. In our example, for instance, transaction T_1^1 that reserved the last available seat, could have triggered another transaction T_3, that changed the value of a variable *full* to true (reflecting that the flight is fully reserved). If there is an integrity constraint in the system that states that if the value of the variable *full* is true, then there are no available seats in the flight. If the compensating transaction for the reservation transaction T_1^1 were to only cancel the reservation, then the consistency of the database will be violated. Therefore, the compensating transaction C_1^1 must also revert the value of $full$ back to false.

Note that in the above example to compensate for the reservation subtransaction T_1^1, the compensating transaction only executed at the site where T_1^1 had executed; that is, at s_1. The reason for this was that the effects of the subtransaction T_1^1 were restricted to only the site s_1. If, however, the effects of T_1^1 had also spread to other sites, then we may need to compensate for T_1^1 at those sites as well. To see this, consider that the variable $full$ in our example is replicated over numerous sites. In that case, the compensating transaction CT_1^1 for the subtransaction T_1^1 will also need to execute at other sites (besides s_1) to change back the value of $full$ to false. This is, however, not very practical since, in general, the compensating transaction for a subtransaction that has committed at one site needs to execute at all the sites. Therefore, we would like to restrict the compensating activity to only the site at which the subtransaction committed. There are at least two ways of achieving this:

1. Prevent any other global transaction from seeing a state written by the subtransaction before its compensating transaction has executed; that is, in our example, we need to ensure that no other global transaction is serialized between T_i^1 and CT_i^1.

2. Restrict global transactions to have no data dependencies between their subtransactions. If a global transaction does not have data dependencies, then its execution at one site is independent of its execution at the other site. In this case, the effects of the committed subtransaction that is to be compensated will not be externalized to other sites and thus compensation can be restricted to the sites at which the transaction committed.

In the remainder of the section, we will assume that either of the above conditions hold and thus compensating transaction for a global subtransaction is restricted to only the site at which the subtransaction executed. As stated above, executing compensating transactions do not result in the standard atomicity of transactions. The resulting notion of atomicity is referred to as *semantic atomicity*.

Definition 6.2 [26]: Let T_i be a global transaction. Let CT_i be a collection of local compensating subtransactions CT_i^1, ..., CT_i^k, one for each site where T_i executes. We say that T_i is *semantically atomic* if and only if either T_i is committed at all sites where it executes, or CT_i^j's are committed at all sites where T_i has committed. □

Since for many the term "transaction" implies full atomicity, the term *saga* [27] has been used to refer to a collection of semantically atomic subtransactions. To ensure semantic atomicity, the GTM must keep a log or record of T_i subtransactions that have been committed. In [41] an *optimistic two phase commit (O2PC)* protocol is introduced to guarantee semantic atomicity. The protocol works as follows.

When a transaction completes, the GTM sends "prepare" messages to the servers at each site, as it is done in the 2PC protocol. However, unlike the 2PC protocol, upon receiving the "prepare" message, the servers optimistically try to commit their subtransactions at that point. The result is reported to the GTM. If all subtransactions committed, then the transaction is declared committed. If not, the transaction is declared aborted, and compensating transactions are run for all the subtransactions that did commit. In the common case where subtransactions are successful, the O2PC lets sites commit sooner than in the 2PC protocol, leading to improved performance. The O2PC protocol was also developed independently in [44]. Processing distributed transactions without an atomic global commit protocol is also studied in [33]. It must be noted that these commit protocols do not require each local DBMS to support a prepared state for commitment of multi-site transactions and are thus attractive for MDBS environments.

We have so far ignored the fact that a transaction that is committed at some sites and aborted at others may violate database consistency. Consider a global transaction T_1 consisting of subtransactions T_1^1 and T_1^2 executing at sites s_1 and s_2 respectively, where T_1^1 is committed and T_1^2 is aborted. It is possible that such a partially committed global transaction may violate inter-site integrity constraints between sites s_1 and s_2. Thus, a compensating transaction CT_1^1 besides performing an inverse of the function performed by T_1^1, must also ensure that after it commits the global constraints between sites s_1 and s_2 hold. Note that even though the execution of the compensating transaction CT_1^1 will reestablish the consistency constraint violated due to the partial commitment of a global transaction, it will not prevent other global transactions that execute at sites s_1 and s_2 before CT_1^1 executes from seeing inconsistent data. Since we require that each transaction sees consistent data, such executions must be prevented. There are two ways in which this can be done.

1. Disallow global inter-site integrity constraints. Note that if no such constraints are allowed in the system, then the above problem will not arise.

2. Prevent any transaction from seeing the effects of both the failed (or compensated-for) and successful subtransactions of the same global transaction. Actually, to prevent such executions, a property of schedules, referred to as *isolation of recovery* (IR) developed in [42], needs to be ensured. Note that if we disallowed global transactions from being serialized in between a subtransaction and its compensating transaction, then the IR property is trivially ensured.

In either of the above cases, ensuring the semantic atomicity of transactions ensures that the effects of the partially committed global transactions have been semantically undone. However, as with the redo and retry approaches, in order to preserve database consistency, besides ensuring semantic atomicity and isolation of recovery (in case there are global inter-site constraints) there is further a need to ensure one of the correctness criteria developed in Section 4 and 5. In [41], a scheme based upon marking sites that ensures isolation of recovery as well as global serializability is developed under an assumption that each local DBMS follows a strict 2PL protocol.

In [50], another protocol that ensures global serializability and isolation of recovery based upon the site graph approach is developed. In order to ensure that the global schedules satisfy the IR property, compensating transactions for the committed subtransactions of the same global transactions are considered as a single global transaction.

Compensation as a technique of recovery was initially introduced in [30]. Schemes based upon compensation were developed in [26, 27, 37] among others. In [27], it is suggested that semantic atomicity can also be useful for dealing with long lived transactions, even in a centralized database system. The long transaction is broken up into subtransactions that commit and release their resources when completed. Long duration transactions are used for many scientific and engineering applications [35]. It is also shown that the log and state information needed for compensation can be stored within the same application database. The notion of sagas is extended in [24] to nested sagas, where a subtransaction may be further decomposed into steps that are compensatable. Other ideas for using semantic atomicity for coping with long lived activities are discussed in [21], [55].

One issue that we have not addressed in this section is that of the design of compensating transaction. Note that some subtransaction may not have simple compensations. For example, say a subtransaction deposits funds in an account. By the time we wish to compensate, the funds may have been withdrawn by another transaction. So a compensation may involve charging the customer a penalty or sending a message to the legal department. Further certain transactions may not be compensatable, e.g., firing of a missile. The design of compensating transactions has been discussed in the literature, e.g., [26, 30, 37].

Compensation mechanisms are closely related to ones that provide multilevel serializability for multilevel transactions [3, 4, 66, 64]. That is, we can view each local database as a complex "object." High level operations can be issued on these objects; they correspond to what we have called subtransactions at a site. High level operations consist of low level "actions" on the internals of the object. A concurrency control mechanism internal to the object ensures that high level operations are atomic. A higher level concurrency control mechanism ensures that operations are interleaved properly.

In the multilevel transaction model each global transaction can be considered as a two level transaction where each local subtransaction is a high level operation that the global transaction applies at a local site. Each local subtransaction consists, in turn, of local read/write operations that the global subtransaction is using to perform a high level global subtransaction operation. Consider, for example, a global transaction that transfers money from account a located at site s_1 to account b located at site s_2. In this case withdrawal money from account a can be considered as one operation that, in turn, consist of reading and writing operation at site s_1.

Local transactions, on the other hand, are considered as one level transactions (i.e. transactions as it is defined in our model here). A concurrent execution of local and global subtransactions at local sites defined as correct if it satisfies a notion of correct execution of multilevel transactions as it is defined in [64]. In such context the reasoning about global serializability and atomicity can be recast into a multilevel transaction model. Such model lets one exploit the semantics of global subtransactions and consequently to relax requirements of global serializability without sacrificing global consistency. Initial work in this direction has been done in [61] but the research in this direction is in its initial stage.

In the multilevel transaction model, a compensation is used to semantically undo results of global subtransactions. Compensation achieves *semantic atomicity* in the following sense. Assume that f^{-1} is the compensating operation of the global subtransaction f. Now consider the execution sequence f followed by f^{-1} such that no other operation (neither another global subtransaction nor any read/write operation of any global and/or local transaction) is executed between f and f^{-1}. Then we require that no subsequent invocation of a subtransaction g could ever detect that both f and f^{-1} were actually executed. That is, g has the same return values, regardless of whether f was actually compensated or neither f nor f^{-1} ever occurred. This condition is stronger than the one we imposed earlier. On the other hand, it permits to use a powerful apparatus of multilevel transaction model for reasoning about global transactions consistency and atomicity. In conclusion of this subsection we conjecture that the compensation conditions of the multilevel transaction model can be replaced by the isolation of recovery condition as we have discussed in our transaction model.

6.5 Combination of the Different Approaches

We have so far described the various approaches that have been studied in the literature for ensuring atomicity of global transactions in a multidatabase system. Each of the approaches has its own merits and demerits. For example, while the redo technique seems attractive since it does not depend upon the semantics of the transactions, its applicability is limited due to the restrictions that need to be imposed upon the data items accessed by global transactions. On the other hand, though the retry and the compensate approaches do not introduce access restrictions, they rely on the semantics of the applications. Further, since not every transaction is retriable or compensatable, their applicability is also limited.

One interesting characteristic that the discussed techniques have is that they are complementary and can thus be supported together in a single system. This enables us to develop a single general solution for ensuring global transaction atomicity such that the system can exploit the good features of each of the developed schemes. To see how the various schemes can be combined and used together we will first need to enhance our global transaction model. A global transaction consists of a set of subtransactions, each of which each is associated with one of the following types:

1. **Compensatable**. A subtransaction is compensatable, if it is possible to undo the effects of the subtransaction by executing a compensating transaction.

2. **Retriable.** Each subtransaction in this class is retriable; that is, if executed from any database state (as long as the database state is consistent) it is guaranteed to commit.

3. **Redoable.** All the other subtransactions that are neither compensatable, nor retriable.

We assume that the GTM has a priori knowledge of which class a particular subtransaction belongs to. For example, this information may be provided by the user. Further, we assume that for each compensatable subtransaction, the user provides a compensating transaction that can be used to undo the effects of the subtransaction. If the user does not specify the type of a certain subtransaction, it is assumed to be a redoable subtransaction by default.

If the global transactions are as specified above, then the GTM can use each of the redo, retry and compensate approach in conjunction. For example, to do so it may follow the global commit protocol below. For the description of the protocol, we refer to a a server on which a compensatable subtransaction executes as a *c*-server. Similarly, a server on which a redoable (retriable) subtransaction executes is referred to as a *rd*-server (*rt*-server). Further, we distinguish one of the subtransactions and refer to it as a *pivot* (the *pivot* subtransaction may be either compensatable, or redoable, or retriable, or neither). The server on which the pivot executes is referred to as the *p*-server.

Consider the following commit protocol that the GTM may use. On the completion of the execution of all the operations of a transaction, the GTM sends a prepare message to each of the servers on the sites at which the transaction executed. On receipt of a prepare message each *rt*-server and each *rd*-server forces the log record it needs to maintain onto stable storage. On the other hand, a *c*-server on receipt of a prepare message, submits the commit operation for the subtransaction to the local DBMS. On receipt of a commit acknowledgment from each of the *c*-servers and an acknowledgment for the prepare message from other cohorts, the GTM submits a commit to the *p*-server. If the pivot is successfully committed, then the transaction will be committed. Note that aborting a transaction may imply that a compensating transaction needs to be scheduled at the sites on which compensatable subtransactions have successfully committed. On receipt of the commit acknowledgment from the p-server, the GTM submits a commit to the remaining servers. If in case a subtransaction is aborted after the pivot has committed (note that the subtransaction must be either a retriable subtransaction or a redoable one), it is either retried or a redo transaction is executed for it depending upon its type.

The above protocol combines each of the scheme that we have discussed for ensuring atomicity of global transactions. Obviously, we assume that each redo subtransaction (except for the pivot) is appropriately restricted and m-serializability of the local schedules is ensured. Similarly, we assume that no other subtransaction of the global transaction depends upon the values of data items read by each retriable subtransaction. The only problem is with regard to the compensatable subtransactions. Recall that for the compensation approach to work, we required that either there be no data dependencies between subtransactions of all (not only the transaction in question) global transactions that execute at a site on which a subtransaction is to be compensated, or we must prevent any other global transaction from seeing the intermediate state before the compensating transaction is committed. Note that requiring that there be no data dependencies between subtransactions of a global transaction may be unnecessarily restrictive. We, therefore, advocate taking the other approach and ensuring that no global transaction is serialized between a subtransaction and its compensating transaction.

It must be noted that it is also possible to further generalize the above protocol and exploit the availability of prepare-to-commit state (if certain sites support such a state) for committing global transactions. The 2PC sites can be sent the prepare message in the first round along with all other servers. Further, a commit decision can be communicated to the 2PC servers along with the commit message to the rd-servers and the rt-servers, after the successful commitment of the pivot subtransaction. Note that it is possible that transactions may not contain subtransactions of one (or more) classes. For example, a given transaction may not have any pivot. It is interesting to note, that if the global subtransaction only consists of compensatable subtransactions, then the above protocol reduces to the O2PC protocol discussed in the compensation section. Similarly, if there are only redo (retriable) subtransactions, then the protocol reduces to the one developed in the redo (retriable) section. Also, if for example each local DBMS supports a prepare-to-commit operation, then the above protocol degenerates to the 2PC protocol.

7 Global Deadlocks

It has been argued [1] that the timeout strategy for dealing with deadlocks performs poorly in a centralized database, as compared to other mechanisms for deadlock detection. However, in a distributed, heterogeneous system it may be attractive because of its simplicity and the independence it gives.

Deadlock detection may also be an option. If sites export wait-for-graph information (see Section 1), then the GTM could run conventional tests to detect cycles in the wait-for-graph. However, autonomous sites may not export this information.

In these cases, it is necessary to devise a strategy for approximating the union of the local wait-for-graphs. The basic idea is that if the GTM has submitted an action of global transaction T_i to a local site s_k, and the GTM has not received a reply, then T_i *could be* involved in a wait at s_k. If another global transaction T_j has executed actions at s_k and has not yet committed everywhere, then T_i could be waiting, directly or indirectly, for T_j. In this fashion the GTM can construct an approximate wait-for-graph: if $T_i \rightarrow T_j$ then T_i could be waiting for T_j. If a deadlock exists, then there will be a cycle in the approximate wait-for-graph. Clearly, the converse is not true; a cycle in the approximate wait-for-graph that is not a real deadlock is called a false deadlock. To reduce the likelihood of false deadlocks, the arc $T_i \rightarrow T_j$ may be added to the approximate wait-for graph only after T_i has been blocked for some threshold amount of time. These ideas are used by the deadlock detection schemes of [8] and [60].

Very little work has been done to determine the performance of deadlock detection or prevention schemes. In particular, it will be important to evaluate the number of false deadlocks that are broken, and to compare detection schemes to simple timeout ones. It is also important to keep in mind that some of the options we have reviewed are deadlock free, mainly the optimistic global controls and strategies where there is no global concurrency control. If timeouts or deadlock detection are not effective, then the deadlock free approaches may be more attractive for a heterogeneous system.

8 Conclusions

Multidatabases are one of the very active database research areas. The 1990 National Science Foundation (USA) Workshop on Future Directions in DBMS Research [59] named the area of multidatabase as one of the two most important research areas for the 90's. In addition, the NSF has sponsored a series of Workshops on Heterogeneous Databases (1989, 1990, 1992). We believe that multidatabase transaction management is of crucial importance if one is to design an effective multidatabase system.

Our work is motivated by a major problem that exists in the contemporary industrial data processing environment—how to manage and guarantee consistency of semantically related data residing in heterogeneous computing environments, distributed over various DBMSs. Since user organizations in a multidatabase system are autonomous and may have substantial capital invested in the DBMS, it is unreasonable to assume that they will be willing to make modifications or lose control over their DBMSs. Therefore, it is imperative to develop methods that do not require major modifications to existing DBMS software but are able to support users data in a consistent and reliable manner.

It would be much easier to develop future multidatabase systems if operating systems, communication interfaces, and database systems were standardized. Although it is utopian to believe that *comprehensive* standards will be developed and enforced, it is nevertheless important to strive for good standards. For example, if TCP/IP, SQL, strict two-phase locking and two-phase commit protocols would be accepted by all vendors, the multidatabase transaction management problem would become much more manageable. We believe that research results in multidatabase systems will provide meaningful input to the standardization effort currently under way.

Multidatabase transaction management research is still at a very early stage and considerably more work needs to be done. In closing, we briefly outline some of major needs we see.

There is a need to study the "low end" of our box spectrum (Section 1). There are many applications where one must deal both with transaction processing boxes and with boxes that do not have a notion of transaction. For example, in a cooperative work environment, some of the data may be stored in a conventional database system, but other may be in file systems, CAD systems, information retrieval systems, etc. How does one work in this environment, without reverting to the lowest common denominator, i.e., without losing transactional capabilities altogether?

There is also a need to understand the performance implications of multidatabase transaction management. Most research to date has focused on *how* to run transactions in a heterogeneous environment, but we also need to evaluate the cost of transaction processing. For instance, how much more expensive will it be to run transactions when each box runs a different concurrency control protocol? In this paper we assumed that the GTM cannot take an advantage of knowledge about mixed types of local DBMSs. For example, if one of the DBMSs is rigorous and another one is strongly serializable, then the GTM assumes that each local DBMS is strongly serializable; the knowledge that one of the DBMS is more restrictive (and, therefore, the GTM could be more permissive) is not used. Availability of such knowledge could possibly increase the concurrency level of global transactions and improve transaction throughput.

Full data consistency and serializability can only be achieved in a multidatabase system by imposing restrictions that many consider severe. Thus, there is a need to identify alternative forms of consistency and ways of restricting "standard" notions of consistency so that positive results can be stated rather than impossibility results. The notions discussed in Section 5 are a start, but other options for correctness include:

1. partitioned notions of consistency — ranging from consistency of a single entity up to database consistency.

2. temporal consistency — for example, the database is consistent each morning at 8am; no promises (or weaker promises) made at other times.

3. degrees of semantic (in)consistency, defined by application-specific predicates.

4. update-based consistency — assume the database is consistent (even if it is not) and apply restrictions to the types of updates that are allowed

ACKNOWLEDGMENTS

The authors are deeply grateful to Sharad Mehrotra for numerous inspiring discussions during the preparation of this paper, and for his help in the actual writing of the paper. The authors are also deeply grateful to Gerhard Weikum, and Hans-J. Schek for inspiring discussions that helped to organize the ideas presented here.

References

[1] R. Agrawal, M. Carey, and L. McVoy. The performance alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, SE-12, 1987.

[2] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. *Data Engineering*, 10(3):5–11, September 1987.

[3] C. Beeri, P. A. Bernstein, N. Goodman, A model for Concurrency in Nested Transaction Systems, *Journal of the ACM*, 36, 2, 1989

[4] C. Beeri, H.-J. Schek, G. Weikum, Multilevel Transaction Management: Theoretical Art or Practical Need, *Proceedings 1st International Conference on Extending Database Technology*, Springer-Verlag Lecture Notes in Computer Science, 303, 1988.

[5] D. Barbara, H. Garcia-Molina, The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems, *Extending Database Technology Conference*, Vienna, 1992.

[6] Y. Breitbart, D. Georgakopolous, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering* 17:9, 1991.

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

[8] Y. Breitbart, W. Litwin, A. Silberschatz. Deadlock Problems in a Multidatabase Environment. *Digest of Papers COMPCON, Spring 91.* San Francisco, CA, pp. 145-151, February 1991.

[9] Y. Breitbart and A. Silberschatz. Multidatabase update issues. *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data*, Chicago, pp. 135–141, 1988.

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

[11] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Transaction management in a failure-prone multidatabase environment. *VLDB Journal*, 1(1), 1992. forthcoming.

[12] Y. Breitbart and A. Silberschatz. Complexity of Global Transaction Management in Multidatabase Systems *Technical Report 198-91* University of Kentucky, November 1991,

[13] Y. Breitbart and A. Silberschatz. Strong Recoverability in Multidatabase Systems. *Proceedings of the Second International Workshop on RIDE/TQP.* Phoenix, AZ, pp. 170-176, 1992

[14] M. Carey, M. Livny. Parallelism and Concurrency Control Performance in Distributed Database Machines, *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data*, Portland, Oregon, pp. 122 - 133, 1989.

[15] Andrew Citron. LU 6.2 Directions. *Proceedings of the International Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1991.

[16] W. Du, A. K. Elmagarmid, W. Kim. Maintaining Quasi Serializability in Multidatabase Systems. *Proceedings of the Seventh International Conference on Data Engineering*, pp. 360-367, 1991.

[17] W. Du and A. K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. *Proceedings of the Fifteenth International Conference on Very Large Databases*, Amsterdam, pp. 347–355, 1989.

[18] K. Eswaran, J. Gray, R. Lorie, I. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19:11, 1976.

[19] A. A. Farrag and M. T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[20] J. N. Gray, M. Anderton, Distributed Computer Systems: Four Case Studies, *Proceedings of the IEEE*, vol. 75, 5, 1987.

[21] D. K. Gifford and J. E. Donahue, "Coordinating Independent Atomic Actions," *Proceedings IEEE COMPCON*, San Francisco, February 1985.

[22] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem Coordinating Multi-Transaction Activities, Technical Report CS-TR-247-90, Department of Computer Science, Princeton University, 1990

[23] H. Garcia-Molina and B. Kogan, Achieving High Availability in Distributed Databases, *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, July 1988, pp. 886-896.

[24] H. Garcia-Molina, K. Salem, D. Gawlick, J. Klein, K. Kleissner, Modeling Long-Running Activities as Nested Sagas, *Database Engineering*, Vol. 14, No. 1, March, 1991.

[25] H. Garcia-Molina. Global Consistency Constraints Considered Harmful for Heterogeneous Database Systems (Position Paper) *Proceedings of the First International Workshop on Research Issues on Data Engineering*. Kyoto, 1991.

[26] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[27] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data*, San Francisco, pp. 249–259, 1987.

[28] V. Gligor, R. Popescu-Zeletin, "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems." Distributed Data Sharing Systems. Eds. F. Schreiber and W. Litwin. North-Holland, 1985, 43-56.

[29] V. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Information Systems*, pp. 287–297, 1986.

[30] J. N. Gray. Notes on database operating systems. *Lecture Notes in Computer Science, Operating Systems: An Advanced Course*, volume 60, pp. 393–481. Springer-Verlag, Berlin, 1978.

[31] J. N. Gray. An Approach to Decentralized Computer Systems, *IEEE Transactions on Software Engineering*, vol 12, 6, 1986.

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

[33] M. Hsu and A. Silberschatz. Unilateral commit: a new paradigm for reliable distributed transaction management. *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, pp. 286–293, April 1991.

[34] D. Johnson, W. Zwaenepoel, Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing, *Journal of Algorithms*, Vol. 11, No. 3, September 1990, pp. 462-491.

[35] H. F. Korth, W. Kim, and F. Bancilhon. On long duration CAD transactions. *Information Sciences*, 46:73–107, October 1988.

[36] J. Klein, Advanced Rule Driven Transaction Management, *IEEE COMPCON*, San Francisco, 1991, pp. 562-567.

[37] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. *Proceedings of the Sixteenth International Conference on Very Large Databases*, Brisbane, pp. 95–106, August 1990.

[38] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, June 1981.

[39] H. F. Korth and G. Speegle. Formal model of correctness without serializability. *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data*, Chicago, pp. 379–388, June 1988.

[40] R. Koo and S. Tueg, Checkpointing and Rollback-Recovery for Distributed Systems, *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January 1987, pp. 23-31.

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

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

[43] N. Lynch. Multi-level atomicity. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[44] P. Muth and T. C. Rakow. Atomic commitment for integrated database systems. *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[45] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. San Diego, Calif., 1992.

[46] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. Ensuring Transaction Atomicity in Multidatabase Systems. *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. San Diego, Calif., 1992.

[47] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Maintaining Database Consistency in Heterogeneous Distributed Database Systems. Technical Report TR-91-04, Department of Computer Science, University of Texas at Austin, 1991.

[48] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, 1991.

[49] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Relaxing serializability in multidatabase systems. *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, Mission Palms, Arizona, 1992.

[50] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A transaction model for multidatabase systems. *Twelfth International Conference on Distributed Computing Systems*, Yokohama, Japan, 1992.

[51] W. Perrizo, J. Rajkumar, and P. Ram. Hydro: A heterogeneous distributed database system. *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data*, Denver, Colorado, pp. 32–39, May 1991.

[52] C. Pu, A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Denver, Colorado, pp. 377-386, May 1991.

[53] C. Pu. Superdatabases for composition of heterogeneous databases. *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, 1988.

[54] Y. Raz. The Principle of Commit Ordering or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers. Technical Report, Digital Equipment Corporation, 1991.

[55] A. Reuter, Contracts: A Means for Extending Control Beyond Transaction Boundaries, Presentation at *Third International Workshop on High Performance Transaction Systems*, Pacific Grove (Asilomar), California, September 1989.

[56] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A strategy for coping with long lived transactions. In *Lecture Notes in Computer Sciences, High performance Transaction Systems*, D. Gawlick, M. Haynie, and A. Reuter, editors, Volume 359, pp. 175–199. Springer-Verlag, 1989.

[57] N. R. Soparkar, H. F. Korth, and A. Silberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer*, December 1991.

[58] L. Sha, J. P. Lehoczky, E. D. Jensen, Modular Concurrency Control and Failure Recovery, *IEEE Transactions on Computers*, Vol. 37, No. 2, February 1988, pp. 146-159.

[59] A. Silberschatz, M. Stonebraker, and J. Ullman. Database systems: Achievements and opportunities. *Communications of the ACM*, October 1991.

[60] P. Scheurermann and H.-L. Tung. A Deadlock Checkpointing Scheme for Multidatabase Systems. *Proceedings of the Second Workshop on RIDE/TQP* Phoenix, AZ pp. 184-191, 1992.

[61] H.-J. Schek, G. Weikum, W. A. Schaad, A Multilevel Transaction Approach to Federated DBMS Transaction Management, *Proceedings of the International Workshop on Interoperability in Multidatabase Systems*, Kyoto, 1991.

[62] Francis Upton IV. OSI Distributed Transaction Processing, An Overview. *Proceedings of the International Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1991.

[63] J. Veijalainen and A. Wolski. Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases. *Proceedings of the 8th International Conference on Data Engineering* Phoenix, AZ 1992.

[64] G. Weikum, Principles and Realization Strategies of Multilevel Transaction Management, *ACM Transactions on Database Systems*, 16, 1, 1991

[65] H. Wachter, A. Reuter, The ConTract Model, *A. K. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications*, Morgan Kaufman, San Mateo, CA, 1992.

[66] G. Weikum, H.-J. Schek. Architectural Issues of Transaction Management in Layered Systems. *Proceedings of the 10th Conference on Very Large Data Bases*. Morgan Kaufmann, Palo Alto, Calif. pp. 454-465. 1984.

[67] A. Wolski and J. Veijalainen. 2PC Agent method: Achieving Serializability in presence of Failures in a Heterogeneous Multidatabase. *Proceedings of the International Conference on Databases Parallel Architectures and their Applications*. pp 321-330, 1990.

[68] K.-L. Wu, P. Yu, C. Pu. Divergence Control for Epsilon-Serializability *Proceeding of the 8th International Conference on Data Engineering*. Phoenix, AZ, February, 1992.