Serializability in Multidatabases

Ramon Lawrence
Department of Computer Science
University of Manitoba
umlawren@cs.umanitoba.ca

December 6, 1996

1 Introduction

Transaction management in a multidatabase environment is extremely costly due to network transmission costs and the handling of replicas. Conventional databases enforce consistency by strict locking and commit protocols which guarantee that the data is always consistent and trustworthy. Implementation of these protocols in a distributed environment using global locking and two-phase commit, provide the necessary consistency at a much greater cost. In a multidatabase environment with cooperating but not necessarily entirely trusting databases, enforcing this consistency across all databases is not only inefficient but also undesirable.

As an alternative to these strict protocols, varying levels of update privileges and consistency constraints are needed which allow the database administrator to determine, at the data item level, the degree of sharing and consistency of the data. There will always be critical data which must be consistent across all sites and therefore require global synchronization protocols, but most data in a multidatabase environment is not of that nature and accesses to it can be less controlled.

The basic problem is that updates in a distributed database environment are expensive[4, 5]. The transmission costs associated with an update transaction are significant even with today's high speed networks, but more importantly, global locking, especially with replicated data, lowers the concurrency at

all sites containing the updated data. Updating a data item replicated across many sites may block hundreds of local or global read transactions at the participating sites. Thus, enforcing serializability reduces overall concurrency.

One alternative is to allow transactions to proceed even while a global update is occurring. The transactions are not guaranteed to get the most recent data; they are only guaranteed to get the most recent committed value at their site. Thus, the newly updated data item becomes available to local or global queries at a given site as soon as that site commits the update, regardless if other sites in the update transaction commit the update. This has the possibility of introducing inconsistencies in the database, but allows the DB more autonomy on how it handles updates and is especially useful when the level of trust between the cooperating DBs is low. Allowing this "unrestricted parallelism" requires a reconciliation mechanism to recompute data item values which become inconsistent.

The common goal of centralized databases where all information is consistent, current and globally agreed upon is unrealistic in a multidatabase environment. A database is an abstraction of a person's or organization's view of the world. Therefore, it is as complicated to reconcile multiple database views of the world as it is for a group of people to agree on their environment. In such a situation, it is not possible to agree on every data item at every point of time, but it is often possible to agree on most things most of the time. Adding or updating information in this environment should not be done atomically. Rather, it falls to the DBMS or database administrator (DBA) to reconcile new information into their particular view of the environment and accept it based on age, structure, and trustworthiness of its source.

Databases participating in a multidatabase environment should be given the autonomy to preserve their view of the world even when this may make them inconsistent with the views of other databases. The concurrency control mechanisms should be flexible enough to allow global locking between trusted databases on critical data items and still permit cascading[14, 11, 10] and voluntary updates on non-critical information.

This paper is organized as follows. Section 2 describes the background work that has been done in the area and an explanation of why current solutions are unacceptable for a real system. A general MDBS architecture common to many of the current solutions is presented in conjunction with appropriate definitions for the MDBS environment. Section 3 defines serializability and justifies relaxing the serializability requirement in this environment. The general MDBS architecture presented in Section 2 is modified in Section 4 into a system which no longer supports global serializability. Section 5 discusses an algorithm for updating independently updatable attributes based on the previous architecture. Finally, future work and concluding remarks are given in Sections 6 and 7.

2 Background Work

According to Breitbart[2], the major task of transaction management in a multidatabase environment is "ensuring the global consistency and freedom from deadlocks of the multidatabase system in the presence of local transactions (i.e. transactions executed outside of the multidatabase system control) and in the face of the inability of local DBMSs to coordinate execution of multidatabase transactions (called global transactions) under the assumption that no design changes are allowed in local DBMSs."

A **multidatabase** is a collection of one or more autonomous databases participating in a global federation for the exchange of data. The basic multidatabase system (MDBS) architecture in Figure 1 consists of a global transaction manager (GTM) which handles the execution of global transactions (GTs) and is responsible for dividing them into subtransactions (STs) for the local database systems (LDBSs) participating in the MDBS. Each LDBS in the system has an associated global transaction server (GTS) which is assumed to convert the subtransactions issued by the GTM into a form usable by the LDBS. Certain algorithms also rely on the GTS for concurrency control and simulating features that the LDBS does not provide. Each LDBS is assumed to be autonomous meaning that no modifications to it are allowed, although many algorithms assume the LDBS to have certain properties. Finally, local transactions not under the control of the GTM

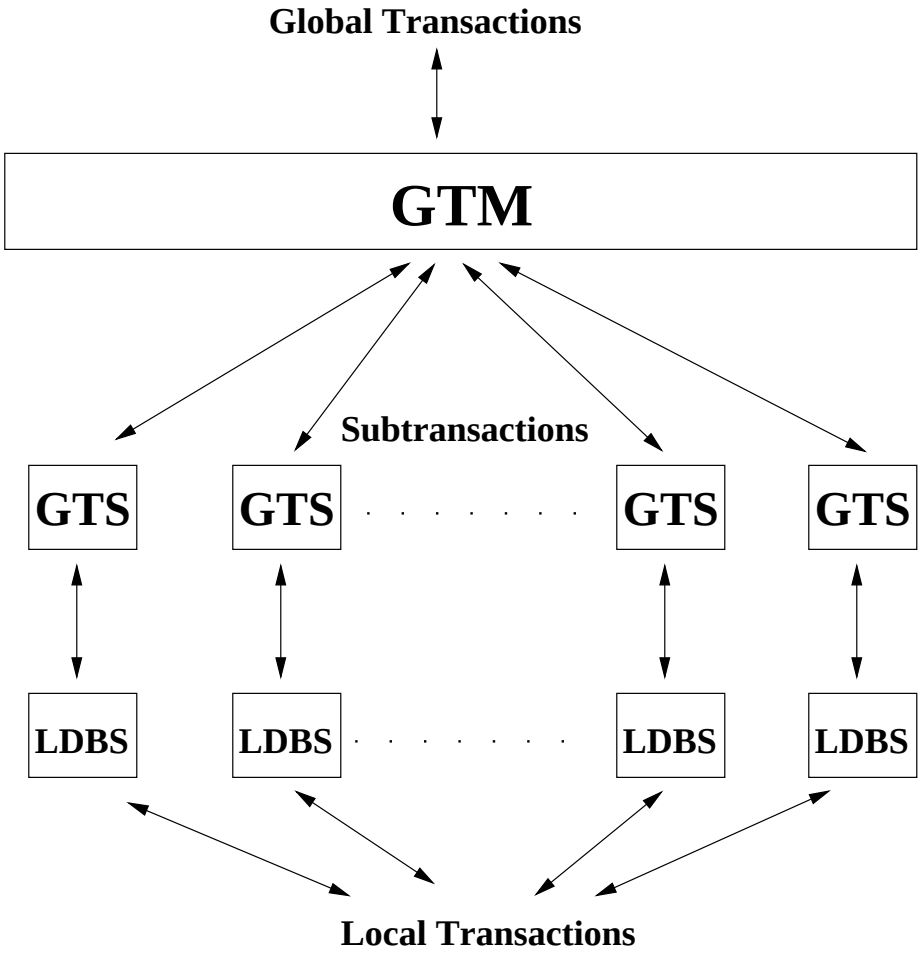

Figure 1: MDBS Architecture

are allowed at each LDBS as if the LDBS was not participating in the MDBS.

Both local and global transactions consist of a finite set of read/write operations to database items and an abort or commit termination condition. Local transactions involve only data stored at one LDBS, while global transactions are allowed to access data at multiple LDBSs. It is generally assumed that each LDBS provides a concurrency control mechanism assuring serializable and deadlock-free schedules. This guarantees local serializability as local transactions and subtransactions of global transactions are serialized at each site. According to Breitbart[3], "a global schedule is **globally serializable** if and only if there exists a total order defined over the committed global transactions that is consistent with the serialization order of committed global transactions at each of the local DBMSs." Basically, there must exist some order of

committing the global transactions such that at each LDBS the subtransactions of the global transactions can be committed in this order. Global serializability can be verified by unioning the local serialization graphs at every site and verifying that the global serialization graph produced is acyclic.

Research in transaction management for multidatabase systems has proceeded in 3 general directions:

- Weakening autonomy by placing restrictions on the local databases
- Enforcing serializability by using local conflicts
- Relaxing serializability constraints by defining alternative notions of correctness

The implications and algorithms for each technique are discussed.

2.1 Weak Autonomy for LDBS

One approach to the transaction management problem is weakening the autonomy restriction on the local databases. Obviously if this weakening is carried out to the extreme, the problem becomes the same as the much simpler problem in distributed databases. There has been some research[12] on the minimal control information of the local databases that can be shared to improve the concurrency of the system, but a more common approach is to assume each local database has certain properties.

One algorithm proposed by Breitbart et al.[2] assumed that each LDBS uses strict-2PL as its concurrency control mechanism. Global serializability is used as a correctness criterion, and the two-phase commit protocol is used for global commits. In this algorithm, the GTM does not commit any subtransactions until all reads and writes of the global transaction are completed at all sites. Using this rule and the fact that all local databases use strict-2PL is enough to guarantee global serializability. Unfortunately, the algorithm is not fault tolerant during global commits. For fault-tolerance, the data items must be divided into two mutually exclusive classes. One class consists of globally updatable items, and the other class contains locally updatable items. A global transaction that updates may write only globally updatable items and may not read locally updatable items. This data item partitioning and restriction on global transactions is also

a common feature of quasi serializability methods. A global deadlock problem is introduced by the global transactions holding locks for their duration and must be detected. The algorithm also needs each database to support the prepare-to-commit state. The **prepare-to-commit state** is reached when the transaction is about to commit but is waiting for an external signal to tell it to commit. Once a transaction is in the prepare-to-commit state, the LDBS cannot abort the transaction.

The algorithm has low concurrency. Since a global transaction holds all its locks at all local sites until all its processing is done at all sites, it may block many other local and global transactions at other sites. This is especially restrictive with long-running or large global transactions that access many sites and may be subject to network delays. Detecting global deadlock is a constant problem and results in significant overhead. Finally, the fault-tolerant version and its partitioning of data items introduces the same problem found with quasi serializability algorithms, which is that this partitioning is difficult and introduces restrictive constraints on local and global transactions. Breitbart does state some other problems with the algorithm including the fact that few commercial databases support the prepare-to-commit state, that it is difficult to know how long to stay in the prepare-to-commit state, and that security issues arise.

According to Breitbart[2], "if all the local DBMSs of a multidatabase system would use the strict two-phase locking protocol and ... the two-phase commit protocol ... then the problem of transaction management in such [a] multidatabase system would be trivially solved, even in the presence of failures." This is a bold statement which holds some validity as the simple algorithm he presented does solve the transaction management problem. Unfortunately, if this algorithm (and many of the others) were implemented on a real system, they would not work because of the very low concurrency. Even in Breitbart's "utopian world" of local database uniformity, the algorithm presented locks too many data items for too long and would not scale well in a production system.

2.2 Enforcing serializability by using local conflicts

Another approach proposed by Georgakopoulos et al.[9, 13, 8] is to enforce serializability by using tickets at each local database. A ticket is a counter used to determine the relative serialization order of the subtransactions at each LDBS. It is effectively a timestamp stored as a data item in the LDBS. Each subtransaction is required to take-a-ticket which corresponds to reading the ticket value and incrementing it. The concurrency control algorithm then allows all subtransactions to proceed at all sites but will only commit a global transaction if the ticket values have the same relative order in all LDBSs.

A detailed description of the algorithm on a global transaction G is:

- The global transaction manager (GTM) sets a timeout and submits the subtransactions (STs) of G to the LDBSs.
- Once all STs are in a prepare-to-commit state at every participating LDBS, validate using a Global Serialization Graph (GSG) test. The nodes of this graph are "recently" committed transactions (no transaction older than the current oldest running transaction), and an edge $G_i \rightarrow G_j$ exists if at least one of the subtransactions of G_i came before (had a smaller ticket than) a subtransaction of G_j in some LDBS.
- The GSG initially contains no cycles. Add a node for G and insert the appropriate edges. If there is no cycle, then G is committed at all sites, otherwise it is aborted and restarted. (G is also aborted if any of its subtransactions fail.)

This method is called the Optimistic Ticket Method (OTM) because it allows transactions to proceed unobstructed in the assumption that no conflicts will arise. OTM guarantees global serializability if:

- LDBSs guarantee local serializability.
- Each global transaction has only one subtransaction per site.
- LDBSs provide primitives to support a prepare-to-commit state.

This method appears very elegant as it requires only one data item per LDBS and preserves the autonomy of the LDBSs. Also, it makes very little assumptions on the power of each LDBS, except for the prepare-to-commit state which is not a common feature of many legacy database systems.

Unfortunately, the algorithm has many hidden problems. If the GTM handles many global transactions, especially large transactions involving multiple sites, the optimistic method may lead to many global transaction conflicts and aborts. There is no method of guaranteeing transaction execution for older transactions so livelock is possible. Using a prepare-to-commit state always introduces the problems of knowing how long to stay in this state and the detection of problems. Another problem is that the ticket at each local site becomes a bottleneck for all global transactions. Every global transaction must access the ticket causing many conflicts and long waiting periods in heavy load situations. The ticket may be locked by a subtransaction for long periods of time as it does not release its lock until all other subtransactions of the global transaction have completed their operations and are in the prepare-to-commit state.

To address the issue of global transaction abort, a Conservative Ticket Method (CTM) was introduced. CTM eliminates global restarts because the transactions are ordered such that it is never necessary to abort a transaction based on a ticket conflict. CTM assumes a relative order that the transactions take tickets in all LDBSs. The algorithm requires a prepare-to-Take-a-Ticket (p-T-a-T) state. A subtransaction enters this state after it completes all its database operations before requesting a ticket and leaves the state when it reads the ticket value. The algorithm is as follows:

- Proceed the same as OTM allowing arbitrarily interleaving at LDBSs until the subtransaction enters the p-T-a-T state.
- The global transactions $G_1, G_2, \ldots, G_n$ are given a relative serialization order.
- The subtransactions of G_{i+1} must wait for all subtransactions of G_i to be in the p-T-a-T state and take a ticket before they can take a ticket.

CTM avoids global transaction rollback but greatly reduces concurrency and results in an almost serial execution of global transactions. Other variations to the ticket method are proposed when the LDBSs guarantee strict schedules or have other properties. One variation[1] assigns timestamps to each global transaction and uses the ticket as a global timestamp storing the timestamp of the last committed global

transaction at that site. A global transaction is aborted if it has a timestamp less than the timestamp at a local database that it is accessing. The algorithm insures that the global transactions are serialized in timestamp order and is also deadlock-free. Unfortunately, it still has the same problems as the basic ticket method including limited concurrency, a hot spot at the ticket item, and the possibility of global transaction aborts. In conclusion, the ticket method is elegant for its simplicity and few restrictions on the LDBSs, but the method is not practical in typical systems due to the concurrency restrictions in some implementations or frequent transaction aborts in other implementations.

2.3 Quasi and Two-Level Serializability

It has been debated that serializability is too strict a correctness criterion especially for multidatabase systems. Quasi serializability[6, 7] is a less restrictive definition of database consistency. An execution is quasi serializable if:

- Local executions are conflict serializable.
- The execution is equivalent to a quasi serial execution in which global transactions are executed sequentially. (i.e. For any 2 global transactions T_i, T_j in the sequential ordering where T_i precedes T_j in the ordering, then all T_i operations precede all T_j operations in all local schedules in which they both appear.)

To determine if a schedule is quasi serializable, construct a quasi serializability graph (QSG) and determine if it is acyclic. The QSG has nodes representing global transactions and an edge $T_i \rightarrow T_j$ iff T_i conflicts (precedes) T_j in some local schedule or some transitive conflict such as $o_i(x), o_1(x), o_1(y), o_2(y)$, . . .,$o_k(t), o_k(z), o_j(z)$ and a local schedule L = $o_i(x), \ldots, o_1(x), \ldots, o_k(t), \ldots, o_k(z), \ldots, o_j(z), \ldots$ exists.

Quasi serializability works by seperating interations between local and global transactions. Local transactions are serializable at each LDBS, and the GTM serializes global transactions. To achieve this separation, the local databases are assumed to be disjoint (have no data items in common), and it is assumed that no data dependencies exist between data items at different sites accessed by the same global transaction.

To remove the data dependencies across sites, the data items are separated into global and local data items and access to them is restricted. There is an algorithm to enforce quasi serializability, but there is no efficient algorithm to determine the data dependence property. The set of schedules that are quasi serializable is a strict superset of the schedules that are conflict serializable. Quasi serializability is not studied in depth because later research yielded two-level serializability (2LSR) which accepts a superset of the schedules accepted by quasi serializability.

Two-level serializability[3] partitions the data into global and local data in attempt to eliminate problems satisfying database constraints. A **database constraint** is an external restriction placed on the data items of the database forcing them to always have certain properties. For example, restricting all bank balances to be positive is a constraint that could be put on a bank database which must always be enforced by the DBMS. Both quasi and two-level serializability try to find alternative ways of satisfying these constraints besides using conflict serializability. There are three types of constraints in a MDBS:

- Local - involve only local data at one site
- Global - involve only global data but may span multiple sites
- Global/Local - involve both local and global data at one site

Each LDBS serializes access to its local data, and the GTM serializes access to global data. The major restriction in 2LSR is that local transactions may not modify global data. This is because global data may be involved in database constraints than span multiple sites (global constraints) and a local transaction has no interaction with these other sites to enforce them. Schedules that are 2LSR preserve all local and global/local database constraints, but global constraints will only be satisfied by imposing rules on transactions like forbidding global transactions from accessing local data. Satisfying all local and global constraints results in a **strongly correct** schedule where the final state is consistent and the state read by each transaction is consistent. Unfortunately, strongly correct executions may not be a strong enough consistency guarantee for some applications.

Two-level serializability has many desirable properties. By separating local and global data, the concurrency of local transactions operating on local data is not directly effected by global transactions (except for sharing LDBS computing power). The GTM can manage the global data efficiently in a manner similar to a distributed database. Two-level serializability may not be suitable for large numbers of databases or partially trusting databases, but it is a promising method for integrating legacy systems in a small environment like a company or department.

The major drawback of 2LSR is implementing the local/global data division. With existing local databases, only global data need be added, but even this may result in data replication and complicated re-engineering. It may not be a trivial task to separate local and global data especially when the number of databases participating in the MDBS is large. Nevertheless, 2LSR is the most of practical of the schemes described so far as it is fairly easily implemented and allows good concurrency.

2.4 Real World Implementations

Multidatabase systems in the real-world tend to be custom systems with little generality. Work in industry in the distributed database field has headed towards providing loose consistency. Loose consistency guarantees the primary copy of the data is correct and propagates the changes to the replicas, which may not be consistent all the time. One system proposed by Sybase[14] provides loose consistency in an open system approach, meaning that no assumptions are made on the type and capabilities of the databases participating in the overall system. The system supports loose consistency by maintaining the primary copies of the data, continually scanning the transaction log to detect updates, and then passing the updates to the replicas. Allowing replicas to be inconsistent for a period of time seems to be the only practical way of implementing a distributed database environment.

3 Serializability and its Problems

The long-held benchmark for defining the correctness of concurrent transaction execution has been serializability. Formally, a schedule is **serializable** if it is conflict equivalent to some serial schedule. Two schedules are **conflict equivalent** if the ordering of any two conflicting operations is the same in both schedules. Two operations **conflict** if they are from two different transactions, operate on the same data item, and one of them is a write. A schedule is **globally serializable** if all local schedules are serializable and there exists an ordering of committing global transactions such that all subtransactions of the global transactions are committed in the same relative order at all sites.

Many very efficient and practical algorithms for enforcing serializability in the centralized environment have been defined such as two-phase locking and timestamping. These algorithms can also be extended to operate in the distributed environment with reasonable additional overhead. The problem of enforcing serializability **efficiently** in a multidatabase environment is still open. It appears unlikely that any efficient algorithm will be found because the multidatabase environment represents a fundamental conceptual shift from the centralized or distributed environments.

The interaction of autonomous databases is the distinguishing factor for a multidatabase. When autonomous entities participate in any form of exchange, there are disagreements, as each entity has its own view and method of working. In a centralized environment, one entity controlled all transactions, so it was fairly easy to guarantee serializability. In a distributed environment, multiple entities control transactions, but they were all working towards the same goal and were willing to compromise their autonomy for a unified system. There is a real-world analogy between a distributed database and a bee-hive. In a bee-hive, each bee has a predefined job for the hive and sacrifices autonomy for the betterment of the hive. Similarly, distributed sites sacrifice their autonomy to the distributed system by sharing control information and cooperating to implement concurrency control and recovery algorithms. In a MDBS, each database has its own methods of concurrency control, data representation, and consistency constraints and is unwilling

to sacrifice them for the "global good." Serializability is hard to enforce because each entity is allowed to proceed in parallel on its own terms. The solutions previously presented enforce global serializability either by restricting the types of transaction access, assuming certain capabilities of the local databases, or by enforcing some global protocol to restrict the actions of the local databases. These methods are infeasible because of their restrictions, high overhead, and low concurrency.

It is questionable that serializability is suitable for the multidatabase environment. With the lack of efficient algorithms despite intense research, the answer appears that serializability is not suitable for a MDBS. The reason why is apparent by examining the real-world analogy between a MDBS and a group of people. A group of people fits the definition of a MDBS perfectly if the word "database" is replaced by the word "person." Each person is an autonomous unit with local data who interacts with other people in the environment to exchange information. In a society, people proceed in parallel with few external restrictions or explicit communication yet meaningful work gets done. It is impossible to serialize these interactions as they are very dependent on timing and independent decisions on different views of the data. This parallelism may involve working on incorrect assumptions or data, doing useless work, or backtracking based on new information, but it is the best method that we as individuals have come up with to get the most done with uncertain and incomplete information. This "human parallelism" involving assumption, parallel work, and backtracking or recomputation to resolve inconsistencies or mistakes may be a useful model for a MDBS.

There are more practical reasons why serializability is hard to support in a MDBS. First, the distances involved in communication and the widely differing processing power of the databases are major issues. Regardless of network speed, there are many messages that must be sent to multiple databases to enforce consistency. If every transaction was instantaneous, then updates would not be overly costly and high concurrency is possible. Unfortunately, network communication slowing transactions and long-running transactions increase the time of transactions, which hold data items longer, and consequently reduce concurrency. Another factor is the number of databases participating in the MDBS may be large. For

example, all the databases on the World Wide Web could be considered a multidatabase if a cooperation protocol was invented. Finally, enforcing serializability assumes that all databases cooperate completely in the MDBS. It weakens local database autonomy by assuming that every update by a global transaction must be accepted regardless of its source or content. Autonomous entities always associate a trust factor with incoming information and may reject it for internal concerns. Current protocols assume totally cooperating databases when in reality, a real-world MDBS would most likely consist of loose federations of databases which exchange information based on their mutual trust. Allowing local databases to control the access to their own data preserves autonomy but violates serializability as each local database can choose to abort or commit a globally committed transaction. This introduces inconsistencies in the global view and brings up the fundamental question of what does a global transaction mean.

4 A MDBS Architecture not supporting Serializability

4.1 General Overview

This section defines a multidatabase environment that does not support serializability based on the analogy of "human parallelism". Specifically, any update by a global transaction can be rejected by any local database without invalidating the global transaction. This results in a global view which may be inconsistent. However, local serializability is preserved and local constraints at each site are satisfied. An optimistic concurrency control mechanism allows local databases to proceed with little intercommunication. This limited synchronization results in temporary inconsistencies and makes it necessary to provide a reconciliation method to make data consistent.

A local database in the MDBS guarantees local consistency and propagates changes to other databases in the MDBS which have the ability to accept or reject these changes. The global view is inconsistent during the update propagation and local information at a given site is inconsistent or stale until the update

propagates to the site. It is the responsibility of the GTS of the local database to reconcile this external update with local information. In the case of conflicts, some sort of transaction semantics must be passed with the update so the GTS can make an informed decision on how this reconciliation can be achieved. This is a hard but necessary problem if the data is not to degenerate into something totally unusable.

The meaning of a global transaction must be redefined for this environment since the global view is inconsistent. Using the real-world analogy of a group of people, a global transaction corresponds to asking the group, or a subset of the group, a question and evaluating the response. Assuming the question is a read-only transaction, the correct answer depends on the user's needs. There are several options including taking the most recent value, the most agreed upon value, or a value recommended by the most trusted person (database). The semantics of an update global transaction is more complex as inconsistent global data may be used to update local data and cause inconsistencies in local data. One method might be to use the local data of the database(s) involved in the update, so that the update does not violate their local constraints. We have no solution at this time, and the solution will probably depend on how strictly the consistency constraints of the local databases are to be enforced. A simple algorithm for write-only transactions (global or local) is presented in Section 6 involving independently updatable attributes. An **independently updatable** attribute can be updated without reading its previous value, and its update has no effect on other values in the database.

4.2 The Consistency/Concurrency Trade-off

Relaxing consistency in a non-serializable MDBS results in greater concurrency and if handled properly, will result in little change in correctness of applications using the database. The consistency/concurrency trade-off arises in all aspects of life with database and computer applications greatly biased to the consistency side. The basic idea is that a lot can be done in parallel if you are willing to make mistakes. A common constraint on all computer systems since their creation is that they be flawless, even though in reality this reliability

is hardly achievable. Database design has always started from the premise that a database is consistent and transactions move the database from one consistent state to another. In fact, a production database is subject to human errors, and the data contained within is often inconsistent, stale, and unnecessary. The DBMS maintains strict serializability of transactions to "preserve" consistency in the system in a more rigorous fashion than external operators validate database information and its usefulness as an abstraction of reality.

In a centralized system, enforcing serializability and consistency is easy using two-phase locking. Most transactions are short, the data is readily available, and maintaining consistency is achievable with little overhead and good concurrency. A distributed system adds the additional overhead of network access and the complexity of distribution, but the algorithms still maintain consistency with good concurrency by taking advantage of the cooperation between the sites. In a MDBS, the concurrency is low because of all the synchronization overhead. The solution is to sacrifice consistency of data in a reasonable way. The level of consistency is highly dependent on the application. Banks for example would want high consistency for bank balances.

Many data items in the MDBS do not need rigorous consistency and others can have the type of transactions on them limited to allow for greater concurrency. Addresses, names, and other vital statistics are slow changing values in the real-world which often have no effect on other data items when changed. These independently updatable attributes are easily kept consistent using update propagation. Other data items, such as a bank balance, can be kept consistent by restricting operations performed on them. For example, addition and subtraction operations always commute, so concurrent updates can proceed in parallel and the "true" consistent value can be obtained at different sites by applying the transaction operations in any order. Studying the types of data items and operations that relaxing consistency is suitable for would be an interesting topic, as would determining how to handle operations that do not commute. Basically, varying levels of consistency should be definable for different data items to allow for increased concurrency. The level of consistency can be defined by the DBA of each LDBS. The consistency level is dependent on

other databases, so this must be defined externally by DBAs of cooperating LDBS and can be enforced by the GTS of each database.

4.3 Architecture Overview

The MDBS architecture is the same as in Figure 1. The exact operation of the MDBS will not be totally defined here as it is specific to the concurrency control algorithm supported. Assumptions needed to support the architecture are presented along with a mechanism to support "federations" between databases in the MDBS efficiently.

The assumptions of the architecture include:

- No violation of LDBS autonomy. (No assumption of control information or concurrency control mechanism.)
- The GTM exists at one site (for simplicity) which can store all needed control information and can be itself considered a database.
- Each LDBS guarantees local serializability and recovery.
- The DBA is responsible for defining the GTS for a given LDBS and adding the additional information to the exported schema needed by the GTM. (This includes defining the trustworthiness of other LDBS participating in the MDBS.)
- A canonical form of data representation agreed upon between the GTS and GTM exists such that the GTM has an integrated global schema of the local schemas. The GTS maps this global conceptual schema into a suitable local schema by converting the subtransactions sent from the GTM into local transactions for the local database.

The GTM has similar functionality as discussed in Section 2. The GTM is responsible for breaking global transactions into subtransactions for the local databases and merging the results returned from the GTSs. The functionality of the GTM is increased because of the possible inconsistencies in the global view. Thus, a mechanism for specifying global transactions must consider timeliness and trustworthiness of data.

Such a mechanism is highly dependent on the concurrency control mechanism chosen and the requirements of the global transactions.

Finally, the definition of a transaction must change. Define the read set (RS) of a transaction as the set of database values read by the transaction. Define the external set (ES) as the set of values used by the transaction for its computation that exist external to the database. A **transaction** T is a program consisting of a sequence of read/write operations, a commit or abort termination operation, a timestamp of its submission, and a computational formulation of its execution sequence such that for every value x written to the database there exists some function f(RS,ES) which determined the value x.

Capturing semantics of transactions is difficult. The previous definition of a transaction is a compromise between full transaction semantics and blind acceptance of updates. If a local database has the ability to reject a global update, it should know as much about the transaction as possible. A function representing the decisions to determine the update may be achievable using compiler technology without the need to define transaction semantics. The function and the transaction timestamp can be used to determine timeliness of data, order of execution, and possible recomputation to reconcile inconsistencies in data. Recomputation is the method humans perform to reconcile new data with existing data. The goal is to achieve parallelism in a MDBS by allowing greater concurrency and using recomputation to fix inconsistencies. The algorithm for updating independent updatable date items does not rely on this recomputation as updating these data items does not effect other database values, but a general concurrency control algorithm must deal with data item dependencies and have a means of reconciling inconsistencies.

4.4 Architecture Discussion

There are several key issues that arise when implementing this architecture. The most complex of which is specifying the level of trust at the data item level that a local database assigns to all other participating databases in the MDBS. This specification is very similar to a federated database system where scalability

is an issue. At some level, trust must be specified as a federation, but it is infeasible for each LDBS to define a distinct export schema for every other LDBS in the MDBS. A solution is possible if we assume that the number of federations in a MDBS is small compared to the number of participating databases. It is highly unlikely that a given database would be "federated" with more than 20 distinct databases in a MDBS which have **update** privileges. The much larger set of read-only access federations and no federations can be grouped into two categories saving needless duplication. A single schema specifying all federations for a given local database can be defined in the GTS as follows:

- The export schema in the GTS is the entire local schema in the LDBS. (Assume mapping from LDBS representation to canonical form of GTS.)
- Each attribute in the export schema has two 32-bit bitmasks representing 32 levels of read/write access. A one in the k-bit ($0 \leq k \leq 31$) position (level) indicates a transaction of k priority can access that attribute. Note that the levels are not hierarchical. A transaction T with access at level k+1 is not allowed to access an attribute with access level k unless T also has access to level k.
- Two levels are special in the bitmasks. Level 0 indicates unrestricted global access. Level 31 indicates no global access is allowed for this attribute. Any other level can be used by DBAs of cooperating databases to specify a federation on that attribute.
- Each transaction is assigned two bitmasks representing its read/write access. A GTS for a given LDBS compares these bitmasks with the attribute bitmasks to determine if the transaction will be accepted by the LDBS.

For example, assume 3 LDBSs decide to share attribute x. The DBAs define the federation by assigning attribute x Level 1 read and write access in the 3 GTSs. Also in each GTS, the 2 other participating LDBSs are defined to have Level 1 access, so any transaction updates originating from these databases are accepted by the others. Thus, a federation on x between these 3 LDBSs is defined and only those databases have access to it. At any time, the federation can be changed by any DBA without modifying the export schema by adding or removing Level access priorities at their site.

This method allows a LDBS to specify a maximum of 30 attribute level federations (extendable by defining a large bitmask). As a general observation, most attributes will have Level 0 (unrestricted global access or read-only global access) or Level 31 (no global access). Since each LDBS will likely participate in a few federations, defining the federations in this manner is easier than defining an export schema for all databases in the MDBS. In terms of expressive power, any size federation can be specified for a given level. For example, it is possible to have 100 LDBSs sharing data item x at a given level. Also, the federations on attributes tend to be clustered. Most attributes are shared between two databases or none at all. It is unlikely that the attributes would need to be partitioned into many different access levels within a federation. Thus, the method should be feasible in a general environment.

Another concern is specifying the read/write access bitmasks for a transaction. If an update is being propagated globally from a local update, then the access priorities are the same as the access priorities for the local database that originated the update. If a global transaction is performing the update, the problem is more complex. The access priority could be defined as the logical OR of the access priorities of all databases which the user has access to. In that case, the GTM must have some mechanism to keep track and verify user access at all databases. For simplicity, each GTS keeps track of all local users for its system which are assigned individual access priorities. A global transaction must be specified as originating from some database. The access priority of this database is then used for the global transaction. Finally, this method is not secure when considering local database security. A user is assumed to have full access to any local database that he can access, which may not be the case. This user is then allowed to access other databases in the federation at a higher priority than deserved. Some mechanism for integrating the two access levels is needed. Accessing the export schema as defined is not intended to be secure if malicious users or databases are in the MDBS but is merely a demonstration of how a schema can be shared without complete global sharing or defining an export schema for each LDBS participating in the MDBS. Protocols implementing security would be interesting future work.

Finally, a new overhead is introduced as each data item has an associated timestamp to determine its timeliness. These timestamps are managed by the GTS as items in the local database and may take up considerable space.

5 Handling Independently Updatable Attributes

As defined previously, an **independently updatable** attribute is a stateless attribute that is not involved in any data dependencies. These attributes can be modified without regard to their previous value, and their modification has no effect on other database values. A method for updating these attributes is as follows:

- For local transactions, their execution is unchanged. On commitment of a local transaction T, the GTS determines the write set of T and assigns the timestamp of T to be the time of commitment. The GTS passes the timestamp, write set, and the local database identifier to the GTM which then distributes the update to sites containing data in the write set.
- For global transactions, the timestamp is the time of submission and the local database identifier is explicitly stated by the user. The write set is extracted from the transaction and passed with the timestamp and local database identifier to the update sites.
- For both local and global transactions, the GTS of a local database has the write set, the timestamp of the transaction, and can calculate the write access of the transaction based on the local database identifier and the privileges assigned by the DBA in the GTS. If the transaction has write access to the attribute(s), for each attribute x in the write set the GTS:
  - Reads and locks x.
  - Reads timestamp, TS, for x.
  - if TS < transaction timestamp then write(x)

The goal is to have each LDBS accept the most-recent value for the data item. By assuming that no data dependencies exist, the problem of updating other data items are eliminated. Interesting future work would be to determine a method of handling data dependencies in general or in the specific case where most databases agree that a given data item is independently updatable but some do not. Notice that global

serializability is violated as a data item may have different values in different databases depending on the timing of transaction execution. The method attempts to serialize transactions in time-based order under the assumption that more recent transactions contain newer information, but does not guarantee that this will always be the case.

6 Future Work

The methods for transaction management defined so far are unacceptable in a real-world environment. A general architecture based on the "human parallelism" model was presented, but it is the very preliminary stages. The feasibility of the architecture still must be justified, and its semantics more precisely defined. Allowing global inconsistencies requires a new and more precise definition of a global transactions and the interaction with local data. It must be shown that a suitable reconciliation algorithm can be defined for attributes with local and global data dependencies and that the period of inconsistency of data is acceptable to applications using the system.

7 Conclusion

An overview of serializability in a MDBS was presented. Current algorithms to enforce serializability have high overhead and low efficiency. The usefulness of serializability as a correctness criterion for multidatabases was debated, and an architecture not enforcing global serializability was introduced. A method for updating independently updatable attributes based on this architecture was given. In conclusion, serializability is hard to support in a MDBS so an alternative method was discussed which worked with on incomplete or stale knowledge and used recomputation to reconcile inconsistencies introduced by the increased parallelism of the local databases.

References

[1] R.K. Batra, M. Rusinkiewicz, and D. Georgakopoulos. A decentralized deadlock-free concurrency control method for multidatabase transactions. In *Proceedings of 12th International Conference on Distributed Computing Systems*, June 1992.

[2] Y. Breitbart. Multidatabase interoperability. *Sigmod Record*, 19(3):53–60, September 1990.

[3] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. Technical Report TR-92-21, University of Texas at Austin, May 1992.

[4] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. In *Database Engineering*, volume 6, 1987. Also published in/as: IEEE CS Technical Com. on Database Engineering Bulletin, Vol.10 No.3, Sep.1987, pp.12–18.

[5] W. Du, A. K. Elmagarmid, W. Kim, and O. Bukhres. Supporting consistent updates in replicated multidatabase systems. *The VLDB Journal*, 2(2):215, April 1993.

[6] W. Du and A.K. Elmagarmid. Maintaining transaction consistency in multidatabases using quasi serializable executions. *IEEE*, pages 152–155, 1991.

[7] W. Du, A.K. Elmagarmid, and W. Kim. Maintaining quasi serializability in multidatabase systems. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 360–367, 1991.

[8] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 286–293, April 1991.

[9] D. Georgakopoulos, M. Rusinkiewicz, and A.P. Sheth. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), February 1994.

[10] T. Mostardi, A. Pelaggi, and C. Siciliano. Achieving consistency of replicated copies with the relay race method. In *Proceedings RIDE-IMS '93. Third International Workshop on Research Issues in Data Engineering*, pages 232–235, 1993.

[11] M.W. Orlowski and Jinli Cao. On optimization of update propagation in multidatabase systems. In *Proceedings TENCON '93. 1993 IEEE Region 10 Conference on Computer, Communication and Power Engineering*, pages 315–318, 1993.

[12] C. Pu. Superdatabases: Transactions across database boundaries. *IEEE Data Engineering*, September 1987.

[13] M. Rusinkiewicz and D. Georgakopoulos. Multidatabase transactions; impediments and opportunities. In *COMPCON Spring '91 Digest of Papers*, pages 137–144, 1991.

[14] Y. Wang and J. Chiao. Data replication in a distributed heteogeneous database environment: an open system approach. In *1994 IEEE 13th Annual International Phoenix Conference on Computers and Communications*, pages 315–321, 315-321.