1989

A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems

Weimin Du

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
89-894

A PARADIGM FOR CONCURRENCY CONTROL IN HETEROGENEOUS DISTRBUTED DATABASE SYSTEMS

Weimin Du
Ahmed K. Elmagarmid

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #894

A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems[1]

Weimin Du and Ahmed K. Elmagarmid
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
(317) 494-1998
ahmed@cs.purdue.edu

[1]This work is supported by a PYI award from NSF under grant IRI-8857952 and grants from AT&T Foundation, Tektronix and Mobil Oil.

Abstract

A heterogeneous distributed database system (HDDBS) is a system which integrates pre-existing databases to support global applications accessing more than one database. This paper outlines possible approaches to concurrency control in HDDBSs. The top-down approach emerges as a viable paradigm for ensuring the proper concurrent execution of global transactions in an HDDBS. The primary contributions of this paper are the general schemes for local concurrency control with pre-specified global serialization orders. Two approaches are outlined. The first is intended for performance enhancement but violates design autonomy, while the second does not violate local autonomy at the cost of generality (it does not apply to all local concurrency control protocols). This study is intended as a guiding light through the maze of concurrency control in this new environment and enormous work remains to be done.

1 Introduction

In a database system, several users may read and update information concurrently. Undesirable situations may arise if the operations of various user transactions are improperly interleaved. Concurrency control is an activity that coordinates concurrently executed operations so that they interfere with each other in an acceptable fashion.

Recently, much attention has been focused on the integration of distributed and autonomous databases. Such an integration results in heterogeneous distributed database systems (HDDBSs). One of the main goals of HDDBSs is to support uniform updates across element databases. A key step in achieving this goal is global concurrency control, which has been widely studied for years [GL84] [GP86] [LS86] [AGS87] [Vid87] [BS88] [EH88] [Pu88].

The problem of concurrency control in HDDBS environments is basically different from that in homogeneous distributed database environments, due to autonomy requirements of local databases. The global concurrency control strategies developed in homogeneous distributed database environments do not work well in HDDBS environments. In addition, most efforts devoted to generalize the classical concurrency control strategies for HDDBSs are not successful. For example, most proposed concurrency control protocols for HDDBSs either violate local autonomy or do not maintain global serializability with the exception of [BS88] (see [DELO89]).

A possible way of doing concurrency control in HDDBSs is to employ unconventional paradigms. One of such paradigms is the value date approach [LT88]. In this paper, we study another unconventional paradigm, top-down approach. In this approach, a global concurrency controller (GCC) is allowed to determine the serialization order of global transactions before submitting them to local sites. This global serialization order is then enforced at all local sites. The paradigm is unconventional in the sense that local concurrency controllers (LCCs) have to control the local executions with pre-specified serialization orders. In this paper, we focus on various methods of performing local concurrency control with pre-specified global serialization orders.

A general discussion of global concurrency control in HDDBSs and the top-down approach is given in Section 2. In section 2, we also outline two general ways of implementing the top-down approach. The first, discussed

in section 3, is intended for high flexibility and performance, at the cost of local autonomy. The second, on the other hand, achieves the global serialization order by controlling the submission of global subtransactions and is discussed in section 4. In section 5, we briefly compare the above two techniques in terms of performance, concurrency degree, local autonomy, and consistency. Finally, some concluding remarks are given in Section 6.

2 Global Concurrency Control in HDDBSs

2.1 Background

A heterogeneous distributed database system is a collection of autonomous but related databases, joined together to support global applications. Each element database (also called a local database system, or LDBS) is controlled by a local database management system (LDBMS). A global database management system (GDBMS) is built on top of these LDBMSs to coordinate executions at local sites. One of the goals of HDDBSs is to provide a uniform access to pre-existing local databases by hiding both the heterogeneity and the autonomy of these LDBSs.

An HDDBS is different from a homogeneous distributed database system in that LDBSs are autonomous. Their having local autonomy means that the LDBSs were designed and implemented independently. It defines the right of an LDBS to make its own decisions about data models and transaction management algorithms, communication with other LDBSs or GDBMS, and the execution of transactions at its site [EV87] [GK88].

In an HDDBS, there are two types of transactions. A local transaction runs on a local site and accesses only local data items. A global transaction, on the other hand, accesses more than one local database. When a global transaction is submitted to an HDDBS, it is decomposed into a set of subtransactions that run at various sites where the referenced data items reside. For a global transaction, it is generally assumed that there is at most one subtransaction at each site [GP86]. At each LDBS, there is an LCC which controls the execution of local transactions and global subtransactions at that site. We assume that these local executions are all serializable. It is the responsibility of the GCC to ensure that the global database consistency is preserved.

Local autonomy has significant effects on global concurrency control in HDDBSs. To maintain global database consistency, a GCC has to detect and resolve all conflicts among global operations. To do this, the GCC needs information about local executions. This information, however, is usually very difficult (if not impossible) to obtain due to local autonomy. An LDBS might not have the information needed by the GCC. Even if it has the information, communication autonomy may prevent the GCC from accessing it. Finally, even with the desired information, the GCC still might not be able to resolve the undesired conflicts because of execution autonomy.

Serializability has been generally used as the correctness criterion in HDDBSs [BS88]. However, it is very difficult to maintain global serializability in HDDBSs. The difficulty results primarily from the local autonomy, as well as the incompatibility between serialization and execution orders of global transactions (see [DELO89]).

In summary, global concurrency control in HDDBSs is difficult. It is particularly difficult to maintain global serializability in these environments. One way of overcoming the difficulties is to use special concurrency control paradigms such as top-down approach of concurrency control (see below).

2.2 The Top-down Approach of Concurrency Control

As we have mentioned, the task of a GCC is to coordinate executions at local sites in such a way that global database consistency is preserved. One way to achieve this is to use a bottom-up strategy. In other words, the serialization order of global transactions is first determined by LCCs. It is the GCC's responsibility to detect and resolve the incompatibilities among local executions. In the bottom-up approach of concurrency control, global subtransactions are usually submitted to LCCs freely and LCCs are also free to execute them in any way they wish. This is the most natural way of doing concurrency control in an autonomous system such as an HDDBS. Some proposals based on this approach can be found in [BS88], [EH88], [LEM88], [Pu88].

A second alternative and the one we shall study in this paper is a top-down approach. In this approach, a GCC determines a global serialization order of global transactions before submitting them to local sites. This order is enforced at all local sites by either LCCs, or the GCC, or both. This approach is unconventional because only those local executions with

the specific serialization order are accepted.

The top-down approach of concurrency control has an important property, i.e., the serializability of global executions. This is because all local executions are serializable and global transactions are serialized in the same way at all local sites. There are two basic steps in a top-down approach of concurrency control: (1) determining the serialization order of global transactions at global level, and (2) achieving the order at local level.

The solution to the first step is quite simple: the GCC can determine the serialization order for global transactions. In order to get good performance, the GCC should choose the order which can be easily achieved at local sites. Intuitively, this order can be estimated according to the order in which global subtransactions are submitted to local sites. We shall not, however, go to details of the problem in this paper.

The solution to the second step is, however, much more complex because most concurrency control protocols do not guarantee any specific serialization order. As we mentioned in [DELO89], the problem of local concurrency control with a specific serialization order is generally very hard and might even be impossible in some cases. The following are two possible solutions to the problem.

- For those HDDBSs which have less restrictive autonomy requirements, local schedulers can be modified in such a way that they guarantee not only the serializability of local executions, but also the specific global serialization order.
- For those HDDBSs whose local schedulers can not be modified, coordinators may be built on top of LCCs to control the submission of global subtransactions so that the given global serialization order is achieved.

The above two approaches are suitable for different applications. In the next two sections, we shall further discuss the feasibility and implementation details of the two approaches respectively.

2.3 Related Works

The idea of determining serialization orders at the global level and then enforcing them locally is not new in HDDBSs. It was first proposed in [GP86]

as one of the conditions under which different local concurrency control mechanisms can be used by the GCC to provide global concurrency control for an HDDBS. The authors of [GP86] realized the necessity of achieving the same serialization order at all local sites. However, it seemed that they did not realize the difficulty of doing this. For example, they believed that if two global subtransactions (belonging to two global transactions respectively) do not share any data, then their serialization order is trivially achieved. This, however, is not true. Local transactions may introduce indirect conflicts between global transactions even if they do not conflict directly (see [DELO89]).

Based on the idea presented in [GP86], Vidyasankar proposed a non-two-phase locking protocol for global concurrency control [Vid87]. His protocol applies to HDDBSs in which LDBSs are interconnected to form a rooted tree. It decomposes a global transaction hierarchically according to the tree structure. The decomposed subtransactions are submitted to local sites atomically. In other words, no subtransactions of another global transaction will be submitted to local sites until the subtransactions of the previous global transaction have all been submitted. As in [GP86], Vidyasankar did not realize the difficulty involved in achieving the serialization order at local sites. In fact, he did not even mention how to achieve the global serialization order at local sites.

Another problem with Vidyasankar's protocol is that it does not take into account the indirect conflicts between global transactions. Therefore, this protocol does not always guarantee global serializability. Another excellent paper based on this approach is [AGS87], which has been reviewed in [DELO89].

3 Modifying Local Schedulers

The first approach to local concurrency control with pre-determined serialization order is characterized by modifying local schedulers. Is the idea of modifying local schedulers feasible? Why is it appropriate? How are various local schedulers actually modified? These are the questions we try to answer in this section.

3.1 Why Modifying Local Schedulers

The main purpose of modifying local schedulers is to guarantee that the serialization order determined by the GCC is ensured at local level. By modifying local schedulers, concurrency control can be done flexibly and, therefore, efficiently. However, the approach is not without problems. The possible objections to the idea of modifying local schedulers might be: (1) it violates local autonomy, and (2) it contradicts the notion that all serializable executions are equivalent. In this subsection, we defend the idea by discussing the two problems in detail.

Before the argument about the first violation, let us investigate how the local autonomy is violated. Local autonomy is generally explained in terms of design, communication, and execution. The design autonomy states that the design decisions of LDBSs such as data models and transaction management protocols are made independently. The modification of local schedulers violates design autonomy because it involves changing these design decisions. Two kinds of design decisions, those of transaction management protocols and transaction processing performance are generally effected.

The violation of local autonomy might be tolerated for the following reasons. First, the requirements of autonomy vary in HDDBS environments. While some have very strong requirements of autonomy, many do not. In other words, most HDDBSs tolerate certain changes of design decisions. Recall that LDBSs are integrated because they are related and wish to cooperate in some way. Therefore, they are usually willing to and sometimes have to compromise some design decisions in order to be integrated into an HDDBS.

The second reason is that local schedulers could be modified in a way that the effects on local users (transactions) are negligible. Recall that the purpose of modifying local schedulers is to ensure that they schedule global subtransactions in the serialization order determined at global level. This usually does not conflict with the basic design decisions made by LDBS designers, such as concurrency control strategy. Only minor implementation details are changed. For example, an LCC based on two-phase locking strategy could be modified such that the two-phase locking rule is not violated (see the next subsection). In addition, such modifications do not necessarily imply bad performance for local transactions. In many modified local schedulers, local transactions are processed in a similar way as in the original local

schedulers. This is possible because in many concurrency control strategies, the serialization order of global subtransactions are determined by the way they, not local transactions, are processed. For example, in a two-phase locking based local scheduler, the serialization order of a subtransaction is totally determined by its lock point.

Another objection to modifying local schedulers is the rejection of some serializable executions. In an ordinary database environment, no specific serialization order is required. The reasons for doing this are (1) it provides a higher degree of concurrency, (2) a specific serialization order does not differ from others in terms of concurrency control, and (3) there is no authority that specifies the serialization order.

In an HDDBS environment, however, the situation is quite different. First, the GCC can determine the global serialization order. This is necessary because the GCC must guarantee that a global serialization order is achieved and it is much easier for the GCC to do this before global subtransactions are submitted. It is also possible because it is the GCC that is really in charge of the serializability of global executions. Second, by specifying a global serialization order, LCCs at different sites can reach the agreement, i.e., the serializability of global execution, more easily. Finally, although local concurrency suffers from pre-specifying global serialization order, it is not as bad as it appears. As we have mentioned, the GCC generally chooses the order based on the order in which global subtransactions are submitted. In most cases, this order is compatible with the orders determined by LCCs. In cases where they are not compatible, the orders determined by LCCs at different sites are usually not compatible, either. As a matter of fact, by pre-specifying a global serialization order, many nonserializable global executions are prevented from occurring at global level.

In summary, the idea of modifying local schedulers violates design autonomy. This violation, however, is tolerable in many HDDBS environments. The idea also contradicts the notion of equal acceptance of serializable executions. But from the practice point of view, the idea is acceptable because it implies potentially better performance and simpler concurrency control.

3.2 How to Modify Local Schedulers: Case Studies

In this subsection, we detail the idea of modifying local schedulers. The discussion is given in the scope of the following three protocols: two phase

locking, timestamp ordering, and serialization graph testing (see [BHG87]). Due to space limitations, we are unable to discuss other interesting protocols. We believe that it is generally possible to modify any concurrency control protocol so that a specific serialization order can be ensured. However, the actual modification and its benefit varies from one protocol to another.

The modifications in this subsection are guided by the following principles:

1. The effects on local transactions should be minimal.
2. The effects on the performance of local executions should be minimal.
3. It should also be simple.

In Addition, we assume that local schedulers can distinguish between local and global operations.

Two-Phase Locking

Two-phase locking (2PL) is the most widely used and probably the simplest concurrency control strategy. In a system using locking, a lock is associated with each data item, and only one transaction can hold the lock at a time. A transaction uses two phase locking if all of its lock acquisitions precede all of its lock releases. In other words, once the transaction has released a lock, it may not acquire any other locks. It has been shown that execution serializability is ensured in a system which verifies that all transactions use two phase locking.

To coordinate the executions of global subtransactions, a data structure, called order stamp, is associated with each data item to record the serialization order of the last global subtransaction that has accessed the data item either directly or indirectly. We say that a global transaction indirectly accesses a data item if this data item has later been accessed by a local transaction which has previously accessed another data item accessed by the global transaction. When a global transaction try to access a data item, its serialization order is compared with the order stamp associated with the data item. The access is permitted (i.e., the lock is granted) only if the serialization order of the global transaction is greater than the order stamp of the data item. In other words, the global serialization order is

achieved by only allowing global subtransactions access a data item in the order determined at global level.

The following is such a modification of the basic two-phase locking protocol.

1. When the scheduler receives a local operation $p_i[x]$ from the transaction manager (TM), it tests if $p_i[x]$ conflicts with another operation $q_j[x]$ which already has the lock. If so, it delays $p_i[x]$. If not, then the scheduler grants the lock to $p_i[x]$, and then sends $p_i[x]$ to the data manager (DM).

   Each time a local transaction gets a lock, the order stamp of that data item is updated. The value to be set is the highest serialization order of global transactions (if any) that have previously accessed a common data item as the local transaction does.

2. When it receives a global operation $p_i[x]$ from the global transaction manager (GTM), the scheduler tests if $p_i[x]$ conflicts with another operation which already has the lock. If not, the scheduler grants the lock to $p_i[x]$, and sends $p_i[x]$ to the DM when the serialization order of T_i is greater than the order stamp associated with x. The associated order stamp of x is accordingly updated to the serialization order of T_i. Otherwise, T_i has to be aborted.

   If $p_i[x]$ does conflict with another operation $q_j[x]$, the scheduler delays $p_i[x]$ if $q_j[x]$ is a local operation, If, on the other hand, $q_j[x]$ is a global operation, $p_i[x]$ will be rejected, forcing the global transaction T_i to abort if the global transaction T_j precedes T_i in the global serialization order. If T_i precedes T_j, then T_j will be forced to abort and $p_i[x]$ takes over the lock.

3. Global operations waiting for a lock are queued based on the global serialization order.

4. Once the scheduler has granted a lock for T_i, it may not release that lock until the DM acknowledges that it has processed the operation that set the lock.

5. Once the scheduler has released a lock for a transaction, it may not subsequently grant any more locks to that transaction (on any data item).

Since all transactions (both local and global) follow two-phase locking rule (rule 5), the modified protocol, M2PL, generates serializable executions only. The global serialization order is also ensured because global subtransactions can only access (either directly or indirectly) a data item in that order. Let us, now informally analyze it with respect to the above three principles.

First, M2PL is similar to 2PL except in cases 2 and 3, where conflicts among global operations are resolved. The change does not effect local operations a lot! Second, and the most important, the performance of the two protocols is almost the same as far as local transactions are concerned. No local operation will be delayed and no local transaction will be forced to abort because of the incompatibility between the local and global serialization order. Finally, although M2PL is more complicated than 2PL, it is still simple enough to be practically implemented.

Great benefit can be obtained by modifying 2PL schedulers. First, a higher degree of concurrency is possible. For example, if two global transactions do not access any common data item, all concurrent executions are possible in M2PL schedulers. In the bottom-up approach, e.g., the protocol proposed in [Pu88], the only possible executions are those in which the order of the two transactions' lock points is compatible with the global serialization order. Second, it also provides better performance. Only those global transactions whose commitment actually causes the nonserializability of the global execution will be aborted.

Timestamp Ordering

In a timestamp ordering (TSO) scheduler, the TM assigns a unique timestamp, $ts(T_i)$, for each transaction, T_i. A TSO scheduler orders conflicting operations according to their timestamps.

> TSO rule: If $p_i[x]$ and $q_j[x]$ are conflicting operations, then the DM processes $p_i[x]$ before $q_j[x]$ if $ts(T_i) < ts(T_j)$.

This rule can be modified as follows.

> Modified TSO rule: If either of the conflicting operations $p_i[x]$ and $q_j[x]$ is a local operation, then the DM processes $p_i[x]$ before

$q_j[x]$ if $ts(T_i) < ts(T_j)$.

else (in this case, both $p_i[x]$ and $q_j[x]$ are global operations) DM processes $p_i[x]$ before $q_j[x]$ if $ts(T_i) < ts(T_j)$ and T_i precedes T_j in the global serialization order, aborts either T_i or T_j otherwise.

Like in the TSO rule, all conflicting operations are ordered according to their timestamps in the modified TSO rule. In addition, the else part of the modified TSO rule guarantees that the conflicting global operations are scheduled in an order compatible with both the global serialization order and their timestamp order.

One problem with the modified TSO rule is that one of conflicting global transactions has to be aborted if their serialization order is incompatible with their timestamp order. This is obviously undesired in applications where conflicts among global operations are not rare. To reduce unnecessary abortions in these environments, the timestamp order of global subtransactions at local level should be compatible with the global serialization order as much as possible. One way of doing this is to submit global transactions to local sites according to the global serialization order. In this case, however, the modified TSO rule works in the same way as the TSO rule does. In other words, the global serialization order can be equally achieved by controlling the submission of global subtransactions in these environments. Therefore, a TSO based local scheduler should be modified only in those environments where conflicts among global operations are rare. In these environments, incompatibilities among TSO order and serialization order do not usually imply abortion of global transactions.

Serialization Graph Testing

A serialization graph testing (SGT) scheduler is the most general scheduler in the sense that it can generate all conflict serializable executions. An SGT scheduler maintains the serialization graph (SG) of the history that represents the local execution it controls. It attains serializable executions by ensuring that the SG always remains acyclic.

The following is a modified basic SGT protocol (MSGT).

When an MSGT scheduler receives an operation $p_i[x]$ from the TM, it first adds a node for T_i to its SG if one does not already

exist. It then adds edges from T_j to T_i for every previously scheduled operation $q_j[x]$ that conflicts with $p_i[x]$. If $p_i[x]$ is a global operation, then it adds an edge from T_j to T_i for every global transaction T_j that precedes T_i in the global serialization order.

1. If the resulting SG contains a cycle, the scheduler rejects $p_i[x]$ and deletes T_i from the SG and all edges incident upon T_i.
2. If the resulting SG is still acyclic, the scheduler accepts $p_i[x]$ and sends it to the DM.

Since the SG is acyclic, the local execution is serializable. In addition, this local serialization order is compatible with the global serialization order. Therefore, the global execution is serializable.

Unlike the 2PL and TSO schedulers, it is usually difficult to achieve a specific serialization order in an SGT scheduler without modifying it. The reason is that it is very difficult to derive the serialization order of local executions at the global level. For example, it might be impossible for the GCC to find out whether two global transactions indirectly conflict with each other, even after they have both completed (see [DELO89]). For those HDDBSs in which some LDBSs use SGT schedulers, the top-down approach with local schedulers being modified is a feasible way (if not the only way) to maintain global serializability.

In summary, the modification of local schedulers is generally possible for any concurrency control protocols. Some protocols (e.g., two phase locking) are more suitable for modification in the sense that significant benefit can be obtained, while others (e.g., timestamp ordering) are not. For some protocols (e.g., serialization graph testing), modification is the only way for a local scheduler to achieve a specific serialization order.

4 Enforcing the Global Serialization Order

In an HDDBS in which the modification of local schedulers is not acceptable, it is still possible to apply the top-down approach to concurrency control. The idea is to coordinate local executions by controlling the submission of global subtransactions. As we mentioned in [DELO89], it is impossible, in

general, to enforce global serializability by using this strategy. For most practical concurrency control protocols, however, it is possible. In this section, we discuss the ways of enforcing global serialization order on those protocols in which it is possible. We also discuss how to deal with protocols in which it is not possible.

4.1 Ensuring Global Serializability

To ensure global serializability, LCCs have to satisfy certain conditions. One sufficient condition is that the LCC should schedule transactions in such a way that, for each transaction, the serialization order is determined by an event occurring in its life time. We call this particular event the serialization event.

Many existing concurrency control protocols satisfy this condition. In 2PL, the serialization order of each transaction is determined by its lock point. In TSO, transactions are serialized according to their time stamps (for more on these orders the reader is referred to [Pu88]). Value date based protocols [LT88] also satisfy this condition. The serialization event of a transaction is simply its value date. All these events occur in the life time of the related transactions.

For those HDDBSs in which all LCCs satisfy this condition, the global serializability can be easily ensured using the top-down approach.

1. The GCC determines the global serialization order for each global transaction before submitting it to local sites.
2. At each local site, global subtransactions are submitted to the LCC according to the global serialization order.
3. A global subtransaction will not be submitted to an LCC until the previously submitted global subtransaction reaches its serialization event.

To guarantee that global subtransactions are submitted to LCCs properly, a coordinator is built on the top of each LCC. The coordinator receives and buffers global subtransactions from the GCC and then submits them to the LCC at proper time.

TSO gives a good example of illustrating the idea. Recall that a TSO scheduler serializes transactions according to their timestamps. It is the co-

ordinator's responsibility to guarantee that global subtransactions get their timestamps in the order pre-determined at global level. Generally, a timestamp is assigned to a transaction at the beginning of its life time. Therefore, what the coordinator needs to do is to submit one global subtransaction at a time to the LCC according to the global serialization order. Once the first has been accepted by the LCC (i.e., a timestamp has been assigned), the next (if any) is submitted.

Global serialization order can be achieved in 2PL schedulers in the same way except that the second subtransaction should not be submitted to the LCC until the first reaches its lock point.

Although global serializability can be achieved in both 2PL and TSO schedulers, the performance may be quite different. Generally speaking, the sooner the serialization event occurs in the life time of a transaction, the higher degree of concurrency could be expected. Therefore, a high degree of concurrency can be obtained in TSO schedulers. This is because that global subtransactions could interleave considerably. In 2PL schedulers, however, only a relatively low degree of concurrency can be obtained. Global subtransactions are almost executed sequentially.

4.2 Ensuring Global Quasi Serializability

There are schedulers that do not satisfy the condition we gave in the previous subsection. For example, it is impossible to enforce a specific serialization order at a site which uses an SGT scheduler. For those HDDBSs in which global serializability cannot be maintained, it might still be possible to apply the top-down approach without modifying local schedulers. In this subsection, we briefly discuss how and under what conditions this can be done.

The basic idea is that, instead of trying to maintain the global serializability, we relax the correctness criterion. Although not appropriate in general, the relaxed criteria should be appropriate in some environments. The criterion that we shall use in this subsection is quasi serializability [DE89]. Quasi serializability theory was introduced in heterogeneous distributed database environments with certain restrictions. A global execution is quasi serializable if it is equivalent to an execution in which all the global transactions are executed sequentially in the same order at all local sites.

The quasi serializability of global executions can be maintained in the top-down way as we did for serializability in the previous subsection. The following protocol is a direct generalization of that for serializability. It could be used to achieve global quasi serializability for those HDDBSS in which global serializability cannot be maintained.

1. A GCC determines an order for global transactions. This order is submitted, along with the global subtransactions, to the local sites.
2. At each local site, the coordinator will submit the global subtransactions according to this global order.
3. The global subtransactions are submitted to the LCC one at a time. The second subtransaction will not be submitted until the first has completed.

A quasi serializable execution might not preserve the consistency of a global database unless the databases meet certain restrictions. The following is a set of restrictions which are sufficient to guarantee the consistency of HDDBSs.

1. No data integrity constraint on data items across LDBSs.
2. No value dependency between subtransactions of the same global transaction [1].

A correctness proof of quasi serializable histories can be found in [DE89]. Following is an example illustrating how transaction consistency of the global database is preserved.

Example 4.1 Consider an HDDBS consisting of two LDBSs, D_1 and D_2, where data items a and b are at D_1, and c, d and e are at D_2. The following global transactions are submitted to the HDDBS:

$$G_1 : w_{g_1}(a)r_{g_1}(d) \qquad G_2 : r_{g_2}(b)r_{g_2}(c)w_{g_2}(e)$$

Let L_1 and L_2 be some local transactions submitted at D_1 and D_2, respectively:

$$L_1 : r_{l_1}(a)w_{l_1}(b) \qquad L_2 : w_{l_2}(d)r_{l_2}(e)$$

[1] A less restrictive condition for value dependency can be found in [ED89].

Let H_1 and H_2 be local histories at D_1 and D_2, respectively:

$H_1 : w_{g_1}(a) r_{l_1}(a) w_{l_1}(b) r_{g_2}(b)$

$H_2 : r_{g_2}(c) w_{l_2}(d) r_{g_1}(d) w_{g_2}(e) r_{l_2}(e)$

Let $H = \{H_1, H_2\}$. Then H is quasi serializable.

Suppose there is no value dependency between the two subtransactions of transaction G_2, the value of data item e written by G_2 at D_2 is not related to the value of data item b read by G_2 at D_1. Therefore, the value of data item e read by local transaction L_2 at D_2 is not related to the value of data item b written by local transaction L_1 at D_1. In other words, there is no relation between L_1 and L_2 (they do not influence each other). The global transactions also interfere with each other properly because they are executed sequentially. Therefore, the transaction consistency of the global database is preserved. □

5 Discussion

The main purpose of a scheduler is to output correct schedules. Other goals include a high degree of concurrency, good performance, as well as a high degree of autonomy in the case of HDDBSs. In this section, we compare the two strategies of doing concurrency control in top-down approach, as well as the bottom-up approach with respect to these issues.

Concurrency

Concurrency is one of the most important measures of the performance of a scheduler. Informally, the concurrency of a scheduler S, denoted $C(S)$, is defined as the set of schedules that can be generated by this scheduler. In an HDDBS, a global schedule is composed of several local schedules. The concurrency of the GCC is, therefore, determined by the concurrencies of local schedulers. Given an HDDBS which consists of n LDBSs whose concurrencies are $C(S_1)$, $C(S_2)$, ..., $C(S_n)$ respectively, the concurrency of the HDDBS is bounded by the subset of $C(S_1) \times C(S_2) \times ... \times C(S_n)$ whose members are globally serializable, where $\times$ stands for Cartesian product. In the following, we shall use $SC(S)$ to denote this serializable subset.

The top-down approach of concurrency control is unable to provide the maximum global concurrency, i.e., $SC(S)$. Since the serialization order has

been determined at global level, only those local schedules which are compatible with the order are possible. For an HDDBS who performs concurrency control by modifying local schedulers, the concurrency of the global scheduler is just the subset of $SC(S)$ whose members are compatible with the pre-determined global serialization order. For example, the schedules output by a modified 2PL schedulers are those that can be output by the original scheduler and compatible with the pre-determined serialization order.

For global schedulers which coordinate local executions by controlling the submission of subtransactions, their concurrency degrees vary according to the concurrency control protocols local schedulers use. Usually, they are low. The reason is that global schedulers have to serialize global subtransactions according to their local serialization events. For most concurrency control protocols, the serialization events lie in the rear part or out of the life time of transactions (TSO is an exception). For a global scheduler based on TSO local schedulers, the concurrency degree is high. For global schedulers based on other kinds of local schedulers, e.g., 2PL, the global subtransactions are executed almost sequentially. These global subtransactions can still interleave with local transactions. The overall concurrency degree is therefore not very bad.

The bottom-up approach of concurrency control is generally good at providing high degrees of concurrency. Since global subtransactions are submitted to LCCs without any control or restriction, all combinations of local schedules are possible as long as they are globally serializable. Theoretically, the concurrency of a bottom-up scheduler could be any subset of $SC(S)$. In practice, however, it is much less. The reason is that it is very hard for the GCC to test whether a global execution is serializable, due to local autonomy.

Other Performance Issues

Besides concurrency, many other issues effect the overall performance of a scheduler. Top-down approach of concurrency control usually provides better performance than bottom-up approach from the points of view of these issues, as explained below.

The first issue that effect the overall performance of a scheduler is deadlock. Unlike in bottom-up approach of concurrency control, there is no global deadlock in top-down approach of concurrency control. Since the serialization order of global transactions has been determined before their

submission to local sites, local schedulers (or coordinators) always serialize them consistently. A global transaction only wait for locks held by those global transactions that precede it in the global serialization order. The situation described in [GP86] will never occur.

In an HDDBS, if global concurrency control is performed top-down by controlling the submission of global subtransactions, local deadlocks are also reduced. An obvious observation is that no local deadlock involving more than one global transaction is possible. A global subtransaction will not get any lock until the previous global subtransaction reaches its lock point.

Another important issue is the abortion of transactions, especially global transactions. In top-down approach of concurrency control, there are fewer global transactions aborted. This is because no global transaction will be aborted because of the inconsistency of local executions. In bottom-up approach of concurrency control, however, local schedulers at different sites may serialize global subtransactions differently, resulting unnecessary abortions. Let us, for example, consider an HDDBS consisting of ten LDBSs. Suppose that a bottom-up approach of concurrency control is employed. In the case where two global transactions arrive the GCC at almost the same time, the possibility that the two global transactions are serialized in the same way at all ten local sites is obviously very low. In other words, it is very likely that one of them has to be aborted. We believe that the abortion of global transactions for the inconsistency among local executions is one of the most important issues that effect the performance of a scheduler.

The idea of performing top-down concurrency control by controlling the submission of global subtransactions as presented in this paper is attractive because coordinators are located at local sites. As a result, communication delay is significantly reduced. The local coordinator will submit the next subtransaction immediately after the previous one passes its serialization event. No communication with the GCC is needed.

Autonomy and Correctness criteria

Autonomy is a very important issue in HDDBS. There is no big difference between top-down and bottom-up approaches as far as autonomy is concerned. Both approaches can be implemented in a way that local autonomy is not violated (e.g., local schedulers are not modified).

In the top-down approach, the idea of controlling the submission of global

subtransactions is obviously preferred if the modification of local schedulers is not allowed. The problem is that it is generally impossible to maintain global serializability, although it is possible in most practical environments. It is, however, possible to maintain quasi serializability in this case, as we have mentioned before. If, on the other hand, the modification of local schedulers is allowed, it is definitely a better idea to do so. In this case, not only global serializability can be maintained generally, but also a high degree of concurrency can be obtained.

The above discussion is summerized in the following table.

	Top-down		Bottom-up
	Modifying LCC	Controlling Submission	
Performance	good	good	bad
Concurrency	good	bad (good for TSO)	good
Autonomy	bad	good	x
Correctness	SR	QSR	SR/QSR

where SR and QSR stand for serializability and quasi serializability, respectively. The "x" in the table means that the evaluation depends on the implementation (e.g., whether local schedulers are modified).

6 Conclusion

In this paper, we have presented a framework for designing concurrency control protocols using top-down approach in two distinctive ways. The first is characterized by the modification of local schedulers and the second controlling the submission of global subtransactions to LCCs. We have illustrated the viability of the first approach by outlining the modifications needed for the 2PL, TSO and SGT strategies. In addition, we gave some justifications on the feasibility and applicability of this method. For those who insist on a high degree of autonomy, we have outlined how global database consistency can be maintained without violating autonomy.

This paper is motivated by the difficulties of doing global concurrency control in traditional ways. We believe that the requirements for local autonomy, performance, and consistency are very different in various HDDBS environments. The ways of doing global concurrency control we presented in this paper can apply to certain environments.

Due to the space limitation and the lack of a proper evaluation model, we are unable to give a thorough and quantitative analysis for the approaches we have presented. We recognize the need for more work on many related issues and each of them are currently being studied in the InterBase project. Our comments regarding performance and deadlock in the paper are judgement calls that need to be confirmed and shall be the topic of future reports.

References

[AGS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. In *IEEE Data Engineering Bulletin*, pages 5–11, September 1987.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems.* Addison-Wesley Publishing Co., 1987.

[BS88] Y. Breitbart and A. Silberschatz. Multibdatabase update issues. In *Proceeding of the International Conference on Management of Data*, pages 135–142, June 1988.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[DELO89] W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. In *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, October 1989.

[ED89] A. Elmagarmid and W. Du. *Supporting Value Dependency for Nested Transactions in InterBase.* Technical Report CSD-TR-885, Purdue University, May 1989.

[EH88] A. Elmagarmid and A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proceedings of the International Conference on Data Engineering*, pages 564–569, 1988.

[EV87] F. Eliassen and J. Veijalainen. Language support for muti-database transactions in a cooperative, autonomous environment. In *TENCON '87, IEEE Regional Conference*, Seoul, 1987.

[GK88] H. Garcia-Molina and B. Kogan. Node autonomy in distributed systems. In *Proceedings of the First International Symposium on Database in Parallel and Distributed Systems*, pages 158–166, 1988.

[GL84] V. Gligor and G. Luckenbaugh. Interconnecting heterogeneous data base management systems. *IEEE Computer*, 17(1):33–43, January 1984.

[GP86] V. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Informtion Systems*, 11(4):287–297, 1986.

[LEM88] Y. Leu, A. Elmagarmid, and D. Mannai. *A transaction management facility for InterBase.* Technical Report TR-88-064, Computer Engineering program, Pennsyvania State University, May 1988.

[LS86] T. Logar and A. Sheth. *Concurrency Control Issues in Heterogeneous Distributed Database Management Systems.* Technical Report, Honeywell Computer Sciences Center, June 1986.

[LT88] W. Litwin and H. Tirri. Flexible concurrency control using value dates. *IEEE Distributed Processing Technical Committee Newsletter*, 10(2):42–49, November 1988.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceeding of the International Conference on Data Engineering*, pages 548–555, February 1988.

[Vid87] K. Vidyasankar. Non-two phase locking protocols for global concurrency control in distributed heterogeneous database systems. In *CIPS Edmonton*, 1987.