The Concurrency Control Problem in Multidatabases: Characteristics and Solutions*

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

Abstract

A *Multidatabase System* (MDBS) is a collection of local database management systems, each of which may follow a different concurrency control protocol. This heterogeneity makes the task of ensuring global serializability in an MDBS environment difficult. In this paper, we reduce the problem of ensuring global serializability to the problem of ensuring serializability in a centralized database system. We identify characteristics of the concurrency control problem in an MDBS environment, and additional requirements on concurrency control schemes for ensuring global serializability. We then develop a range of concurrency control schemes that ensure global serializability in an MDBS environment, and at the same time meet the requirements. Finally, we study the tradeoffs between the complexities of the various schemes and the degree of concurrency provided by each of them.

1 Introduction

The problem of transaction management in a multidatabase system (MDBS) has received considerable attention from the database community in recent years [BS88, Pu88, BST90, ED90, GRS91, MRKS91, SKS91]. The basic problem is to design an effective and efficient transaction management scheme that allows users to access and update data items managed by pre-existing and autonomous local database management systems (DBMSs) located at different sites. Transactions in an MDBS are of two types:

- **Local transactions.** Those transactions that execute at a single site.
- **Global transactions.** Those transactions that may execute at several sites.

The execution of global transactions is co-ordinated by the *global transaction manager* (GTM) – a software package built on top of the existing DBMSs whose function is to ensure that the concurrent execution of local and global transactions is serializable. Ensuring global serializability in an MDBS is complicated by the fact that each of the participating local DBMSs is a pre-existing database system whose software cannot be modified. As a result,

- Each local DBMS may follow a different concurrency control protocol.
- Local DBMSs may not communicate any information (e.g., conflict graph) relating to concurrency control to the GTM.
- The GTM is unaware of indirect conflicts between global transactions due to the execution of local transactions at the local DBMSs. This is due to the fact that pre-existing local applications make calls to the local DBMS interfaces, and thus the GTM, which is built on top of the local DBMSs, is not involved in the execution of the local transactions.

Various schemes to ensure global serializability in an MDBS environment have been previously proposed (e.g., [BS88, Pu88, ED90, GRS91]). These proposed schemes have been ad-hoc in nature, and no analysis of their performance, the degree of concurrency provided by them, or their complexity has been made. In this paper, we provide a unifying framework for the study and development of concurrency control schemes in an MDBS environment. We utilize a notion similar to *serialization events* [ED90] (referred to as *O-elements* in

*This material is based in part upon work supported by NSF grants IRI-8805215 and IRI-9003341 and by grants from the IBM corporation and the NEC corporation. This material is based in part upon work supported by the Center for Manufacturing and Robotics of the University of Kentucky and by NSF Grant IRI-8904932.

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

‡Department of Computer Sciences, University of Kentucky, Lexington, KY 40506

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0288...$1.50

[Pu88]) in order to reduce the problem of ensuring global serializability in an MDBS to the problem of ensuring serializability in a centralized DBMS. We then develop a range of concurrency control schemes for ensuring global serializability in an MDBS environment. Finally, we compare the degree of concurrency provided by each of the various schemes and analyze their complexities.

2 MDBS Concurrency Control

In this section, we show how the problem of ensuring global serializability in an MDBS can be reduced to the problem of ensuring serializability in a centralized DBMS. Since centralized concurrency control is a well studied problem and a number of schemes for ensuring serializability in centralized DBMSs have been proposed in the literature, the development of concurrency control schemes for MDBSs is thus simplified. We begin by first discussing the MDBS model.

2.1 The MDBS Model

An MDBS is a collection of pre-existing DBMSs located at sites $s_1, s_2, \ldots, s_m$. A transaction T_i in an MDBS environment is a totally ordered set of **read** (denoted by r_i), **write** (denoted by w_i), **begin** (denoted by b_i) and **commit** (denoted by c_i) operations (a global transaction may have multiple begin and commit operations, one for each site at which it executes). A *global schedule* S is the set of all operations belonging to local and global transactions with a partial order $\prec_S$ on them. The *local schedule* at a site s_k, denoted by S_k, is the set of all operations (belonging to local and global transactions) that execute at s_k with a total order $\prec_{S_k}$ on them. The schedule S_k is a *restriction*[1] of the global schedule S.

We assume that the GTM is centrally located, and controls the execution of global transactions. It communicates with the various local DBMSs by means of *server* processes (one per transaction per site) that execute at each site on top of the local DBMSs. We assume that the interface between the servers and the local DBMSs provides for operations to be submitted by the servers to the local DBMSs, and the local DBMSs to acknowledge the completion of operations to the servers. The local DBMSs do not distinguish between local transactions and global subtransactions executing at its site. In addition, each of the local DBMSs ensures that local schedules are serializable[2].

[1] A set P_1 with a partial order $\prec_{P_1}$ on its elements is a *restriction* of a set P_2 with a partial order $\prec_{P_2}$ on its elements if $P_1 \subseteq P_2$, and for all $e_1, e_2 \in P_1$, $e_1 \prec_{P_1} e_2$ if and only if $e_1 \prec_{P_2} e_2$.

[2] In this paper, we limit ourselves to conflict serializability (CSR) [Pap86], which we shall refer to, in the remainder of the paper, as serializability.

2.2 Serialization Functions

In order to develop our idea, we need to first introduce the notion of *serialization functions*, which is similar to the notion of serialization events [ED90]. Let τ_k be the set of all global subtransactions in S_k. A serialization function for s_k, *ser*, is a function that maps every transaction in τ_k to one of its operations such that for any pair of transactions $T_i, T_j \in \tau_k$, if T_i is serialized before T_j in S_k, then $ser(T_i) \prec_{S_k} ser(T_j)$.

For example, if the *timestamp ordering* (TO) concurrency control protocol is used at site s_k, and the local DBMS at site s_k assigns timestamps to transactions when they begin execution, then the function that maps every transaction $T_i \in \tau_k$ to T_i's begin operation is a serialization function for s_k.

For a site s_k, there may be multiple serialization functions. For example, if the local DBMS at s_k follows the *two-phase locking* (2PL) protocol, then a possible serialization function for s_k maps every transaction $T_i \in \tau_k$ to the operation that results in T_i obtaining its last lock. Alternatively, the function that maps every transaction $T_i \in \tau_k$ to the operation that results in T_i releasing its first lock is also a serialization function for s_k[3].

Unfortunately, serialization functions may not exist for sites following certain protocols (e.g., *serialization graph testing* (SGT)). For such sites, serialization functions can be introduced using *external* means by forcing conflicts between transactions [GRS91]. For example, every transaction in τ_k can be forced to write a particular data item at site s_k, say, *ticket*. Thus, if some transaction $T_i \in \tau_k$ is serialized before another transaction $T_j \in \tau_k$ in S_k, then T_i must have written *ticket* before T_j wrote it. Thus, the function that maps every transaction $T_i \in \tau_k$ to its write operation on *ticket* is a serialization function for s_k. We denote by ser_k, any one of the possible serialization functions for site s_k.

2.3 Global Serializability

Serialization functions can be used to ensure global serializability in an MDBS environment. In the following theorem, we state a sufficient condition for ensuring global serializability in an MDBS environment.

Theorem 1: Consider an MDBS where each local schedule is serializable. Global serializability is assured if there exists a total order on the global transactions such that at each site s_k, for all pairs of global transactions G_i, G_j executing at site s_k, if $ser_k(G_i) \prec_{S_k} ser_k(G_j)$, then G_i is before G_j in the total order. □

[3] Actually, any function that maps every transaction $T_i \in \tau_k$ to one of its operations that executes between the time T_i obtains its last lock and the time it releases its first lock is a serialization function for s_k.

For every global transaction G_i, we define transaction $\widehat{G}_i$ to be a restriction of G_i consisting of all the operations in $\{ser_k(G_i) : G_i \text{ executes at site } s_k\}$. For global schedule S, we define schedule $ser(S)$ to be the set of operations belonging to all transactions $\widehat{G}_i$, with a partial order on them. Further, $ser(S)$ is a restriction of S. In the global schedule S, two operations conflict if both access the same data item and one of them is a write operation. However, the notion of conflict between operations in $ser(S)$ is defined differently. Operations $ser_k(G_i)$ and $ser_l(G_j)$ conflict in $ser(S)$ if and only if $k = l$. From Theorem 1, it follows that S is serializable if $ser(S)$ is serializable.

Theorem 2: Consider an MDBS where each local schedule is serializable. A global schedule S is serializable if $ser(S)$ is serializable. □

We have thus reduced the problem of ensuring serializability in an MDBS environment to the problem of ensuring that $ser(S)$ is serializable. Since global transactions execute under the control of the GTM, the GTM can control the execution of the operations in $ser(S)$ in order to ensure that $ser(S)$ is serializable. Thus, for ensuring global serializability in an MDBS environment, we can restrict ourselves to the development of schemes for ensuring that $ser(S)$ is serializable.

To do so, we split the GTM into two components, GTM_1 and GTM_2 (see Figure 1). GTM_1 utilizes the information on serialization functions for various sites in order to determine for every global transaction G_i, operations $ser_k(G_i)$, and submits them to GTM_2 for processing. The remaining global transaction operations (that are not $ser_k(G_i)$) are directly submitted to the local DBMSs (through the servers). Further, GTM_1 does not submit an operation belonging to a global transaction G_i (except the first operation) to either the local DBMSs or GTM_2 unless an acknowledgement for the completion of the execution of the previous operation belonging to G_i (at the local DBMSs) has been received.

GTM_2 is responsible for ensuring that the operations submitted to it by GTM_1 execute at the local DBMSs in such a manner that $ser(S)$ is serializable. Our concern, for the remainder of the paper, shall be the development of concurrency control schemes for GTM_2 that ensure $ser(S)$ is serializable.

3 Characteristics of the Concurrency Control Problem

A number of schemes for ensuring serializability in centralized DBMSs exist in the literature (e.g., 2PL, TO, SGT). Any one of them can be employed by GTM_2 in order to ensure that $ser(S)$ is serializable. However, certain characteristics of MDBS environments make some of the existing schemes unsuitable for

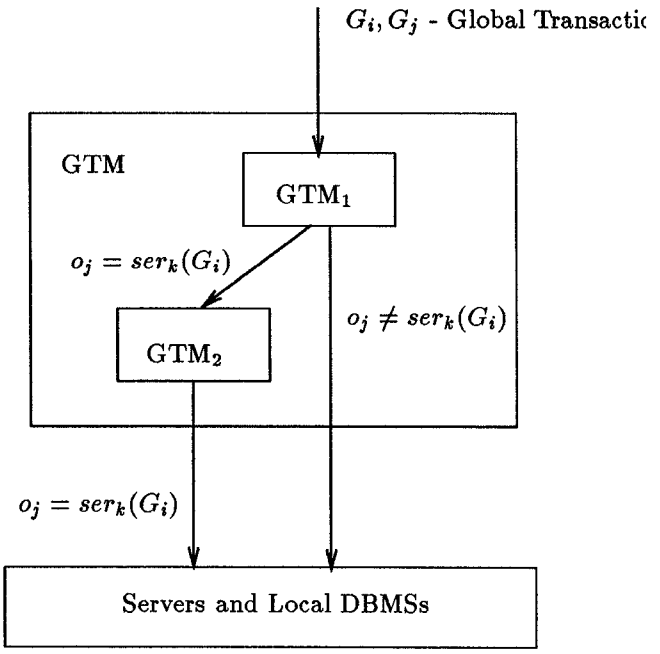

Figure 1: The GTM Components

ensuring the serializability of $ser(S)$. Below, we list some of the factors that play an important role in the design of concurrency control protocols for ensuring the serializability of $ser(S)$.

1. In most MDBS environments, we expect the number of sites to be small in comparison to the number of *active* global transactions in the system (that is, global transactions that have begun execution, but have not yet completed execution). Thus, since any pair of operations $ser_k(G_i)$ and $ser_k(G_j)$ conflict in $ser(S)$, $ser(S)$ may contain a large number of conflicting operations. As a result, if, for example, the 2PL protocol were used to ensure the serializability of $ser(S)$, then there would be frequent deadlocks; if instead, the TO or optimistic protocols were used, a large number of transaction aborts would result. An abort of transaction $\widehat{G}_i$ in $ser(S)$ corresponds to the abortion of the global transaction G_i, which may be expensive, and thus highly undesirable in an MDBS environment. Thus, protocols to ensure serializability of $ser(S)$ must avoid transaction aborts, that is, they must be *conservative* (e.g., conservative 2PL, conservative TO [BHG87]). This is quite feasible in an MDBS environment.

2. Concurrency control protocols that provide a low degree of concurrency may be unsuitable for ensuring that $ser(S)$ is serializable since such protocols may cause a number of operations in $ser(S)$ to be delayed unnecessarily. Such delays may adversely affect the performance of the system since unnecessarily delay-

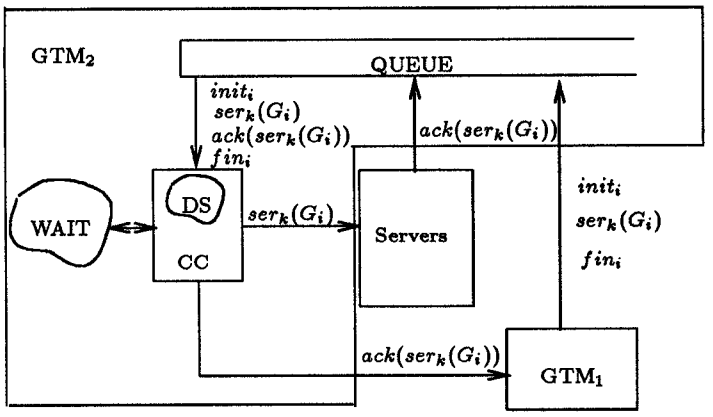

Figure 2: Basic Structure of GTM_2

ing an operation in $ser(S)$ may correspond to delaying the execution of an entire global subtransaction. For example, for a site s_k that uses the TO scheme, ser_k may map each transaction to its begin operation. As a result, causing an operation of $ser(S)$ to wait could cause the execution of an entire global subtransaction to be delayed.

3. A common problem with concurrency control protocols that provide a high degree of concurrency is that they incur substantial overhead for scheduling a single operation (e.g., SGT). However, it may be justifiable to use such concurrency control schemes with high overhead in order to ensure the serializability of $ser(S)$, since the overhead involved in scheduling an operation of $ser(S)$ is amortized, not over one operation, but over all the operations belonging to the corresponding global subtransaction. Thus, the gain in terms of increased throughput, faster response times, and the number of global subtransactions that may be permitted to execute concurrently by a concurrency control scheme that permits a high degree of concurrency may outweigh the overhead associated with the concurrency control scheme.

The above factors imply that concurrency control schemes for ensuring the serializability of $ser(S)$ must be conservative, and must provide high degrees of concurrency (even though they may involve a high overhead). Conservative schemes for ensuring global serializability in MDBS environments have been proposed in [BS88, ED90], while non-conservative schemes have been proposed in [Pu88, GRS91].

4 Structure and Complexity of Conservative Schemes

In this section, we describe the basic structure of conservative concurrency control schemes employed by GTM_2, and the methodology we adopt for analyzing their complexity. As mentioned earlier, GTM_1 submits the $ser_k(G_i)$ operations belonging to each global transaction G_i (or alternatively, each transaction $\widehat{G}_i$) to GTM_2. GTM_1 inserts these operations into a queue, QUEUE. In addition, for every transaction $\widehat{G}_i$, GTM_1 inserts into QUEUE, the operations $init_i$ and fin_i (whose utility is discussed below). We now briefly describe the operations in QUEUE for an arbitrary transaction $\widehat{G}_i$ and site s_k.

- $\boldsymbol{init_i}$: This operation is inserted into QUEUE by GTM_1 before any other operation belonging to $\widehat{G}_i$ is inserted into QUEUE.
- $\boldsymbol{ser_k(G_i)}$: This operation is inserted into QUEUE by GTM_1 in order to request the execution of operation $ser_k(G_i)$.
- $\boldsymbol{ack(ser_k(G_i))}$: This operation is inserted into QUEUE by the servers when the local DBMSs complete executing operation $ser_k(G_i)$.
- $\boldsymbol{fin_i}$: This operation is inserted into QUEUE by GTM_1 after $ack(ser_k(G_i))$, for all $ser_k(G_i) \in \widehat{G}_i$ have been received by GTM_1.

Note that the $init_i$ and fin_i operations do not belong to transaction $\widehat{G}_i$.

In Figure 2, we present the basic structure of GTM_2. CC is any conservative concurrency control scheme for ensuring serializability of $ser(S)$. CC selects operations from the front of QUEUE, in order to *process* them. Associated with CC are certain data structures (DS) that are manipulated every time an operation selected from QUEUE is processed by it. In addition, the following actions are performed by CC when it processes an operation o_j in QUEUE.

- $\boldsymbol{init_i}$: Operation $init_i$ contains information relating to transaction $\widehat{G}_i$ (e.g., the operations in $\widehat{G}_i$, the set of sites G_i executes at). This information is utilized by CC to determine conflicting operations and is added to DS.
- $\boldsymbol{ser_k(G_i)}$: Operation $ser_k(G_i)$ is submitted to the local DBMSs for execution (through the servers).
- $\boldsymbol{ack(ser_k(G_i))}$: Operation $ack(ser_k(G_i))$ is forwarded to GTM_1.
- $\boldsymbol{fin_i}$: Information relating to $\widehat{G}_i$ is deleted from DS.

We denote by $act(o_j)$, the actions performed by CC when it processes an operation o_j in QUEUE. An essential feature of conservative schemes is that they ensure serializability without resorting to transaction aborts. As a result, it may not always be possible for CC to process an operation when it is selected from QUEUE since processing every operation when it is selected from QUEUE may result in non-serializable schedules.

```
procedure Basic_Scheme():
Initialize data structures;
while (true)
begin
Select operation o_j from the front of QUEUE;
if cond(o_j) then begin
                    act(o_j);
                     while (there exists an operation
                  o_l ∈ WAIT such that cond(o_l) is true)
                        begin
                        act(o_l);
                        WAIT := WAIT -{o_l}
                        end
                    end
             else WAIT := WAIT ∪ {o_j};
end
```

Figure 3: Basic Structure of Conservative Schemes

Thus, associated with every operation o_j in QUEUE, is a condition, $cond(o_j)$, that is defined over DS and that must hold if o_j is to be processed by CC. If $cond(o_j)$ does not hold when operation o_j is selected from QUEUE by CC, then o_j is added to a set of waiting operations, WAIT, to be processed at a later time when $cond(o_j)$ becomes true. Thus, every conservative scheme for ensuring the serializability of $ser(S)$ has the same basic structure as shown in Figure 3. However, different conservative schemes differ in the values for $act(o_j)$ and $cond(o_j)$ for the various operations, and the data structures associated with the scheme. Thus, a conservative concurrency control scheme can be specified by specifying $cond(o_j)$, $act(o_j)$ for the various operations, and the data structures maintained by the scheme.

We now illustrate, by an example, how our abstraction for the structure of conservative schemes can be used to specify a simple scheme similar to the conservative TO scheme [BHG87], which we refer to as Scheme 0. The data structures maintained by Scheme 0 consist of queues (initially empty), one associated with every site s_k. For an operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ are defined as follows.

- $cond(init_i)$: *true*.
- $act(init_i)$: Every operation $ser_k(G_i)$ is inserted at the end of the queue for site s_k.
- $cond(ser_k(G_i))$: Operation $ser_k(G_i)$ is the first operation in the queue for site s_k.
- $act(ser_k(G_i))$: Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.
- $cond(ack(ser_k(G_i)))$: *true*.
- $act(ack(ser_k(G_i)))$: Operation $ser_k(G_i)$ is dequeued from the front of the queue for s_k, and $ack(ser_k(G_i))$ is sent to GTM_1.
- $cond(fin_i)$: *true*.

No actions are performed by Scheme 0 when a fin_i operation is processed. Since transactions $\widehat{G}_i$ are serialized in the order in which the $init_i$ operations are processed, trivially, Scheme 0 ensures that $ser(S)$ is serializable.

We now analyze the complexity of CC which has the basic structure as shown in Figure 3. The *complexity* of CC is the average number of steps it takes CC to schedule a transaction $\widehat{G}_i$. For the purpose of analyzing the complexity of the various schemes, we assume the following.

- Every transaction $\widehat{G}_i$ has, on an average, d_{av} operations (that is, the average number of sites at which a global transaction executes is d_{av}).
- At no point during the execution of CC does the difference between the number of $init_i$ and fin_i operations processed by CC exceed n.

Since every transaction $\widehat{G}_i$ is assumed to contain d_{av} operations, the average number of steps taken by CC to schedule $\widehat{G}_i$ is the sum of:

- The number of steps required by CC to process $init_i$,
- $d_{av}\times$(the number of steps required by CC to process $ser_k(G_i)$),
- $d_{av}\times$(the number of steps required by CC to process $ack(ser_k(G_i))$), and
- The number of steps required by CC to process fin_i.

Note that every time an operation o_j is processed by CC (that is, $act(o_j)$ is executed), operations $o_l \in$ WAIT for which $cond(o_l)$ holds are processed, too. Thus, the number of steps required by CC to process an operation o_j is the sum of:

- The number of steps in $cond(o_j)$,
- The number of steps in $act(o_j)$, and
- The number of steps required to determine the operations $o_l \in$ WAIT for which $cond(o_l)$ holds due to the execution of $act(o_j)$.

Scheme 0 can be shown to have complexity $O(d_{av})$. Since checking if an operation $ser_k(G_i)$ is the first operation in the queue for site s_k takes $O(1)$ time, the number of steps in $cond(o_j)$, for all o_j, is $O(1)$.

Further, since enqueueing and dequeueing an operation takes $O(1)$ time, and d_{av} operations are enqueued when an $init_i$ operation is processed, the number of steps in $act(init_i)$ is $O(d_{av})$, and the number of steps in $act(o_j)$, $o_j \neq init_i$, is $O(1)$. Since $cond(init_i)$ and $cond(ack(ser_k(G_i))$ are both *true*, the only operations in WAIT are $ser_k(G_i)$, for some site s_k and transaction $\widehat{G}_i$. Further, since execution of $act(o_j)$ can cause $cond(ser_k(G_i))$ to hold only if $o_j = ack(ser_k(G_l))$, for some transaction G_l, and $ser_k(G_i)$ follows $ser_k(G_l)$ in the queue for s_k, the number of steps required to determine the operations $o_l \in$ WAIT for which $cond(o_l)$ holds due to the execution of $act(o_j)$, for all operations o_j, is $O(1)$. Thus, the complexity of Scheme 0 is $O(d_{av})$.

In Scheme 0, when $init_i$ is processed, the processing of every operation $ser_k(G_i) \in \widehat{G}_i$ is restricted to follow the processing of operations ahead of $ser_k(G_i)$ in the queue for s_k. No restrictions on the processing of operations are added when $ser_k(G_i)$ or $ack(ser_k(G_i))$ is processed. We refer to such schemes in which restrictions on the processing of $ser_k(G_i)$ operations are added to DS only when an $init_i$ operation is processed, as *begin transaction schemes* or *BT-schemes*. The schemes proposed [BS88, ED90] are BT-schemes. On the other hand, a scheme in which restrictions on the processing of $ser_k(G_i)$ operations are added every time an $init_i$ or a $ser_k(G_i)$ operation is processed is referred to as an *operation scheme* or *O-scheme*. O-schemes, in general, result in a *higher degree of concurrency* than BT-schemes (a concurrency control scheme, say CC_1, is said to provide a higher degree of concurrency than another concurrency control scheme CC_2 if, for any given order of insertion of operations into QUEUE by GTM_1, CC_2 does not cause a fewer number of operations to be added to WAIT than CC_1). In this paper, we present an O-scheme that permits the set of all serializable schedules. Even though BT-schemes cannot provide a high degree of concurrency, certain BT-schemes (e.g., Scheme 0) are attractive, since they have low complexities compared to O-schemes.

In the following sections, we present two BT-schemes, and an O-scheme. The schemes ensure that $ser(S)$ is serializable. We specify the concurrency control schemes by specifying the data structures maintained by the scheme, and $cond(o_j)$, $act(o_j)$ for the various operations. We also state the complexity of each of the schemes, and compare the degree of concurrency provided by the various schemes. A detailed analysis of the complexity of the schemes and proofs of their correctness can be found in [MRB+91].

5 The Transaction-site Graph Scheme

Even though Scheme 0 has a low complexity, $O(d_{av})$, it has a serious drawback in that it permits a low degree of concurrency. Below we present a scheme, which we refer to as Scheme 1, and that provides a higher degree of concurrency than Scheme 0. It utilizes a data structure similar to the site graph introduced in [BS88], which we refer to as the *transaction-site graph* (TSG). A TSG is an undirected bi-partite graph consisting of a set of nodes V corresponding to sites (site nodes) and transactions in $ser(S)$ (transaction nodes), and a set of edges E. Site and transaction nodes are labeled by the corresponding sites and transactions, respectively. Edges in the TSG may be present only between transaction nodes and site nodes. An edge between a transaction node $\widehat{G}_i$ and a site node s_k is present only if operation $ser_k(G_i) \in \widehat{G}_i$, and is denoted by either $(s_k, \widehat{G}_i)$ or $(\widehat{G}_i, s_k)$. The set of edges $\{(\widehat{G}_i, s_k) : ser_k(G_i) \in \widehat{G}_i\}$ are referred to as $\widehat{G}_i$'s edges.

Associated with every site s_k, are two queues : an *insert queue* and a *delete queue*. Initially, all queues are empty, and for the TSG, both $V = \emptyset$ and $E = \emptyset$. Processing of certain operations in the insert queues is constrained by "marking" them. For an operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ are defined as follows:

- $cond(init_i)$: *true*.
- $act(init_i)$: $\widehat{G}_i$ and its edges are inserted into the TSG. Also, for every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is inserted at the end of the insert queue for site s_k. If the TSG contains a cycle involving edge $(\widehat{G}_i, s_k)$, then operation $ser_k(G_i)$ in the insert queue for site s_k is marked.
- $cond(ser_k(G_i))$: For every transaction $\widehat{G}_j$ such that $ser_k(G_j) \in \widehat{G}_j$, if $act(ser_k(G_j))$ has executed, then $act(ack(ser_k(G_j)))$ has also completed execution. In addition, if $ser_k(G_i)$ is marked, then it is the first element in the insert queue for site s_k.
- $act(ser_k(G_i))$: Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.
- $cond(ack(ser_k(G_i)))$: *true*.
- $act(ack(ser_k(G_i)))$: Operation $ser_k(G_i)$ is deleted from the insert queue for site s_k (note that $ser_k(G_i)$ may not be at the front of the insert queue for site s_k), and it is added to the end of the delete queue for site s_k. Operation $ack(ser_k(G_i))$ is sent to GTM_1.
- $cond(fin_i)$: For every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is the first element in the delete queue for site s_k.
- $act(fin_i)$: $\widehat{G}_i$ and its edges are deleted from the TSG. For every operation $ser_k(G_i) \in \widehat{G}_i$, $ser_k(G_i)$ is deleted from the delete queue for site s_k.

Scheme 1 permits the TSG to contain cycles, but prevents cycles in the serialization graph of $ser(S)$ by marking operations whose processing may potentially lead to cycles in the serialization graph. Further, processing of a marked operation is delayed until all the operations ahead of it in the insert queue have been processed (note that processing of unmarked operations is not constrained in any way).

Theorem 3: Scheme 1 ensures that $ser(S)$ is serializable. □

The number of steps required to detect cycles in the TSG dominates the complexity of Scheme 1. Cycles in the TSG can be detected using *depth-first search* [AHU74]. Note that the TSG has at most $m+n$ nodes and at most nd_{av} edges.

Theorem 4: The complexity of Scheme 1 is $O(m+n+nd_{av})$. □

6 The Transaction-site Graph-with-dependencies Scheme

Scheme 1, presented in the previous section, does not exploit the knowledge of the order in which operations are processed (the TSG is checked only for cycles). As a result, Scheme 1 places unnecessary restrictions on the processing of operations. The transaction-site graph-with-dependencies scheme, referred to in the sequel as Scheme 2, is presented below and exploits the knowledge of the order in which operations are processed. In order to permit schedules not permitted by Scheme 1, Scheme 2 utilizes a structure similar to the TSG. The structure contains, in addition to transaction and site nodes, *dependencies* (denoted by $\rightarrow$) between edges incident on a common site node, and is referred to as *Transaction Site Graph with Dependencies* (TSGD). A TSGD is a 3-tuple (V, E, D), where V is the set of transaction and site nodes, E is the set of edges and D is the set of dependencies. Dependencies specify the relative order in which operations are processed and are used to restrict the processing of operations. If $(\widehat{G}_i, s_k)$ and $(s_k, \widehat{G}_j)$ are edges in the TSGD, then a dependency of the form $(\widehat{G}_i, s_k) \rightarrow (s_k, \widehat{G}_j)$ denotes that $ser_k(G_i)$ is processed before $ser_k(G_j)$.

Acyclicity of the TSGD plays an important role in ensuring that $ser(S)$ is serializable. Below, we formally state the conditions under which a TSGD is said to be *acyclic*. Consider a TSGD containing edges (v_1, v_2), (v_2, v_3), ..., (v_k, v_1), $k > 2$. This set of edges form a *cycle* if $v_i \neq v_j$, for all $i, j = 1, 2, \ldots, k$, $i \neq j$, and either one of the following is true.

- For all i, $i = 2, 3, \ldots, k$, dependency $(v_{i-1}, v_i) \rightarrow (v_i, v_{(i \bmod k)+1}) \notin D$.
- For all i, $i = 2, 3, \ldots, k$, dependency $(v_{(i \bmod k)+1}, v_i) \rightarrow (v_i, v_{i-1}) \notin D$.

We say the TSGD is acyclic if it does not contain any cycles. We now specify, for every operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ that preserve the acyclicity of the TSGD. Initially, for the TSGD, $V = \emptyset$, $E = \emptyset$, $D = \emptyset$.

- ***cond*(*init*$_i$)**: *true*.
- ***act*(*init*$_i$)**: $\widehat{G}_i$ and its edges are inserted into the TSGD. For every operation $ser_k(G_i) \in \widehat{G}_i$, for all transactions $\widehat{G}_j \in V$ such that $ser_k(G_j) \in \widehat{G}_j$ and $act(ser_k(G_j))$ has executed, dependencies $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i)$ are added to D. The set of dependencies, D, is further modified as follows.

procedure Eliminate_Cycles($(V, E, D), \widehat{G}_i$):

1. Mark all edges "unused". For all transactions $\widehat{G}_j \in V$, $s\_par(\widehat{G}_j) := null$, $t\_par(\widehat{G}_j) := null$. Also, $v := \widehat{G}_i$, $\Delta := \emptyset$.
2. If for all pairs of distinct edges $(v, u), (u, w)$, either
   - $w \neq \widehat{G}_i$ and (u, w) is marked "used", or
   - there is a dependency $(v, u) \rightarrow (u, w)$ in $D \cup \Delta$, or
   - $head(s\_par(v)) = u$.

   then go to step (4).
3. Choose a pair of distinct edges $(v, u), (u, w)$ such that
   - $w = \widehat{G}_i$ or (u, w) is not marked "used", and
   - there is no dependency $(v, u) \rightarrow (u, w)$ in $D \cup \Delta$, and
   - $head(s\_par(v)) \neq u$.

   Mark (u, w) "used". If $w = \widehat{G}_i$, then add to Δ the dependency $(v, u) \rightarrow (u, \widehat{G}_i)$. If $w \neq \widehat{G}_i$, then $s\_par(w) := u \circ s\_par(w)$, $t\_par(w) := v \circ t\_par(w)$, and $v := w$. Go to step (2).
4. If $v \neq \widehat{G}_i$, then $temp := head(t\_par(v))$, $t\_par(v) := tail(t\_par(v))$, $s\_par(v) := tail(s\_par(v))$, $v := temp$, go to step (2).
5. return(Δ).

Figure 4: The procedure Eliminate_Cycles

$$D := D \cup \text{Eliminate\_Cycles}((V, E, D), \widehat{G}_i)$$

The procedure Eliminate_Cycles (specified in Figure 4) returns a set of dependencies Δ such that $(V, E, D \cup \Delta)$ does not contain any cycles involving $\widehat{G}_i$.

- ***cond***$(ser_k(G_i))$: For all transactions $\widehat{G}_j \in V$, if dependency $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i) \in D$, then $act(ack(ser_k(G_j)))$ has completed execution.
- ***act***$(ser_k(G_i))$: For every transaction $\widehat{G}_j \in V$ such that $ser_k(G_j) \in \widehat{G}_j$ and $act(ser_k(G_j))$ has not yet been executed, dependencies $(\widehat{G}_i, s_k) \rightarrow (s_k, \widehat{G}_j)$ are added to D. Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.
- ***cond***$(ack(ser_k(G_i)))$: *true*.
- ***act***$(ack(ser_k(G_i)))$: Operation $ack(ser_k(G_i))$ is sent to GTM_1.
- ***cond***(fin_i): For every operation $ser_k(G_i) \in \widehat{G}_i$, there does not exist a $\widehat{G}_j \in V$ such that $(\widehat{G}_j, s_k) \rightarrow (s_k, \widehat{G}_i) \in D$.
- ***act***(fin_i): $\widehat{G}_i$, along with its edges and dependencies is deleted from the TSGD.

We now discuss the procedure Eliminate_Cycles that takes as arguments the TSGD and a transaction $\widehat{G}_i \in V$. Eliminate_Cycles exploits the knowledge of the order of in which operations are processed and returns a set of dependencies Δ with the following properties.

- Every dependency in Δ is of the form $(\widehat{G}_j, s_k) \rightarrow (\widehat{G}_i, s_k)$, for some transaction $\widehat{G}_j \in V$ and some site $s_k \in V$.
- In $(V, E, D \cup \Delta)$ there are no cycles involving $\widehat{G}_i$.

Eliminate_Cycles attempts to detect cycles involving $\widehat{G}_i$ in the TSGD, and then eliminates them by adding dependencies to Δ. It traverses edges in the TSGD "marking" them as it goes along so that an edge is not traversed multiple times. If an edge incident on $\widehat{G}_i$ is traversed, then Eliminate_Cycles concludes that there is a cycle involving $\widehat{G}_i$ and adds appropriate dependencies to Δ in order to eliminate the cycle.

In Eliminate_Cycles, v is the current transaction node being visited (site nodes are not visited). Unlike depth-first search[AHU74], in Eliminate_Cycles, a transaction node may be visited multiple times. For a transaction node $\widehat{G}_j$, $t\_par(\widehat{G}_j)$ stores the list of transaction nodes to which backtracking from $\widehat{G}_j$ must take place, and $s\_par(\widehat{G}_j)$, the list of site nodes from which $\widehat{G}_j$ is visited, every time it is visited. Functions *head*, *tail* and $\circ$ are as defined for lists[4].

Eliminate_Cycles returns a set of dependencies Δ such that $(V, E, D \cup \Delta)$ contains no cycles involving $\widehat{G}_i$. Since Eliminate_Cycles is invoked every time an $init_i$ operation is processed, it can be shown by a simple induction argument on the number of $init_i$ operations processed, that the TSGD is always acyclic, and thus $ser(S)$ is serializable.

Theorem 5: Scheme 2 ensures that $ser(S)$ is serializable. □

The number of steps in Eliminate_Cycles dominates the complexity of Scheme 2. It can be shown that Eliminate_Cycles terminates in $O(n^2 d_{av})$ steps.

Theorem 6: The complexity of Scheme 2 is $O(n^2 d_{av})$. □

Scheme 2 provides a higher degree of concurrency than Scheme 0. However, Scheme 2 does not provide a higher degree of concurrency than Scheme 1 since certain dependencies in the set of dependencies Δ returned by Eliminate_Cycles may be unnecessary for the purpose of ensuring that $(V, E, D \cup \Delta)$ contains no cycles involving $\widehat{G}_i$. Thus, there may exist a set of dependencies, Δ_1, such that $\Delta_1 \subset \Delta$ and $(V, E, D \cup \Delta_1)$ does not contain a cycle involving $\widehat{G}_i$.

We formally define the notion of *minimality* as follows. A set of dependencies Δ is minimal with respect to the TSGD and a transaction $\widehat{G}_i \in V$ iff

- $(V, E, D \cup \Delta)$ does not contain any cycles involving $\widehat{G}_i$, and
- for all $d \in \Delta$, $(V, E, D \cup \Delta - d)$ contains a cycle involving $\widehat{G}_i$.

The set of dependencies Δ that are returned by Eliminate_Cycles may not be minimal with respect to (V, E, D) and $\widehat{G}_i$, and thus unnecessary restrictions may be imposed on the processing of operations, hurting the degree of concurrency. In order to impose minimal restrictions on the processing of operations and to provide maximal concurrency without jeopardizing the serializability of $ser(S)$, Δ must be minimal with respect to (V, E, D) and $\widehat{G}_i$. However, the problem of computing such a Δ is NP-hard [GJ79], and is a consequence of the following NP-completeness result.

Theorem 7: The following problem is NP-complete. Given a TSGD and a transaction node $\widehat{G}_i \in V$. Is $\Delta = \emptyset$

[4] For a list $l = [l_1, l_2, \ldots, l_p]$ and element l_0, $head(l)$ returns l_1, $tail(l)$ returns $[l_2, \ldots, l_p]$, and $l_0 \circ l$ returns $[l_0, l_1, l_2, \ldots, l_p]$.

not minimal with respect to the TSGD and transaction $\widehat{G}_i$? □

7 An O-Scheme that Permits all Serializable Schedules

The problem with the BT-schemes presented in the previous sections is that they either provide a low degree of concurrency or have high complexity. This is due to the requirement that all the restrictions on the processing of $\widehat{G}_i$'s operations in order to ensure serializability of $ser(S)$ be added to DS when $init_i$ is processed (since no restrictions are added when $ser_k(G_i)$ operations are processed). Thus, BT-schemes cannot provide a very high degree of concurrency since they *a priori* restrict the processing of operations to permit only a subset of serializable schedules. Furthermore, BT-schemes that attempt to provide even a moderately high degree of concurrency are intractable, as shown in the previous section.

In this section, we present an O-scheme that permits the set of all serializable schedules, which we refer to as Scheme 3. Scheme 3 adds restrictions on the processing of $\widehat{G}_j$'s operations to DS every time an $init_i$ or $ser_k(G_i)$ operation is processed. As a result, when an $init_i$ or a $ser_k(G_i)$ operation is processed, Scheme 3 only adds minimum restrictions to DS such that processing the next $ser_k(G_i)$ operation cannot cause $ser(S)$ to be non-serializable (additional restrictions are added when the next $ser_k(G_i)$ operation is processed). Since, at any point, minimum restrictions are imposed on the processing of operations in order to ensure serializability of $ser(S)$, Scheme 3 permits the set of all possible serializable schedules. Further, the computation of the minimum restrictions to be added to DS every time an operation is processed is not too difficult, and Scheme 3 can be shown to have a complexity $O(n^2 d_{av})$.

In scheme 3, associated with every transaction $\widehat{G}_i$ is a set $ser\_bef(\widehat{G}_i)$ of transactions such that if $\widehat{G}_j \in ser\_bef(\widehat{G}_i)$, then $\widehat{G}_j$ is serialized before $\widehat{G}_i$ in $ser(S)$. Also, at any point p during the execution of Scheme 3, for every site s_k,

- $last_k$ is the transaction $\widehat{G}_i$ that is the last among transactions in $\{\widehat{G}_j : ser_k(G_j) \in \widehat{G}_j\}$ to have executed $act(ser_k(G_i))$ before point p.
- set_k is the set of transactions $\{\widehat{G}_j : (ser_k(G_j) \in \widehat{G}_j) \wedge (act(init_j)$ has executed before $p) \wedge (\ act(ser_k(G_j))$ has not executed before $p)\}$.

Initially, for all s_k, $last_k = null$, $set_k = \emptyset$, and for all $\widehat{G}_i$, $ser\_bef(\widehat{G}_i) = \emptyset$. For an operation o_j in QUEUE, $cond(o_j)$ and $act(o_j)$ are defined as follows.

- ***cond*($init_i$)**: *true*.
- ***act*($init_i$)**: For every operation $ser_k(G_i) \in \widehat{G}_i$, $\widehat{G}_i$ is added to set_k. The set $ser\_bef(\widehat{G}_i)$ is updated as follows to include all the transactions serialized before $\widehat{G}_i$.

$$ser\_bef(\widehat{G}_i) := \cup_{ser_k(G_i) \in \widehat{G}_i \wedge last_k \neq null} (ser\_bef(last_k) \cup \{last_k\}).$$

- ***cond*($ser_k(G_i)$)**: $ser\_bef(\widehat{G}_i) \cap (set_k - \{\widehat{G}_i\}) = \emptyset$. If $last_k = \widehat{G}_j$, then $act(ack(ser_k(G_j)))$ has executed.
- ***act*($ser_k(G_i)$)**: $\widehat{G}_i$ is deleted from set_k and $last_k$ is set to $\widehat{G}_i$. Since for all transactions $\widehat{G}_j \in set_k$, $ser_k(G_j)$ has not been processed when $ser_k(G_i)$ is processed, $ser_k(G_i)$ executes before $ser_k(G_j)$ executes, and $\widehat{G}_i$ is thus serialized before $\widehat{G}_j$ in $ser(S)$. Thus, for certain transactions $\widehat{G}_j$, $ser\_bef(\widehat{G}_j)$ is updated as follows to include all the transactions serialized before $\widehat{G}_j$.

  Let $Set_1 = (ser\_bef(\widehat{G}_i) \cup \{\widehat{G}_i\})$, $Set_2 = \{\widehat{G}_l : ser\_bef(\widehat{G}_l) \cap set_k \neq \emptyset\}$. For all transactions $\widehat{G}_j$ such that either $\widehat{G}_j \in set_k$, or $\widehat{G}_j \in Set_2$,

  $$ser\_bef(\widehat{G}_j) := ser\_bef(\widehat{G}_j) \cup Set_1.$$

  Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- ***cond*($ack(ser_k(G_i))$)**: *true*.
- ***act*($ack(ser_k(G_i))$)**: Operation $ack(ser_k(G_i))$ is sent to GTM_1.
- ***cond*(fin_i)**: $ser\_bef(\widehat{G}_i) = \emptyset$.
- ***act*(fin_i)**: For all transactions $\widehat{G}_j$ such that $\widehat{G}_i \in ser\_bef(\widehat{G}_j)$, $\widehat{G}_i$ is deleted from $ser\_bef(\widehat{G}_j)$. For all $ser_k(G_i) \in \widehat{G}_i$ such that $last_k = \widehat{G}_i$, $last_k$ is set to *null*.

Scheme 3 ensures that for all transactions $\widehat{G}_i$, $\widehat{G}_i \notin ser\_bef(\widehat{G}_i)$, and thus, $\widehat{G}_i$ is never serialized before itself.

Theorem 8: Scheme 3 ensures that $ser(S)$ is serializable. □

The complexity of Scheme 3 is dominated by the number of steps required in order to update $ser\_bef(\widehat{G}_j)$, for certain transactions $\widehat{G}_j$, when $act(ser_k(G_i))$ executes.

Theorem 9: The complexity of Scheme 3 is $O(n^2 d_{av})$. □

In [MRB+91], we have shown that at any point during the execution of Scheme 3, if for some set set_k, $set_k \neq \emptyset$, then for some transaction $\widehat{G}_i \in set_k$, $ser_k(G_i)$ is processed by Scheme 3. Further Scheme 3 permits the set of all serializable schedules, that is, if $ser_k(G_i)$ operations are inserted into QUEUE by GTM_1 in such a manner that processing every $ser_k(G_i)$ operation when it is selected from QUEUE results in a serializable schedule, then Scheme 3 does not add any $ser_k(G_i)$ operation to WAIT. Thus, Scheme 3 permits a higher degree of concurrency than Schemes 0, 1 and 2.

8 Conclusion

There has been no systematic study of the concurrency control problem in MDBS environments. Existing schemes for ensuring global serializability in MDBSs are ad-hoc, and no analysis of their performance, the degree of concurrency provided by them, or their complexity has been made. In this paper, we have reduced the problem of developing schemes for ensuring global serializability in an MDBS environment to that of developing conservative schemes for ensuring serializability in a centralized DBMS. Since concurrency control in centralized DBMSs is a well studied problem, the development of concurrency control schemes for MDBSs is simplified.

We have proposed a model for conservative concurrency control schemes, and have developed a number of conservative schemes for ensuring serializability, including one that permits the set of all serializable schedules. We have analyzed the complexities of each of the developed schemes and compared the degree of concurrency provided by the various schemes. Since conservative schemes delay the execution of operations belonging to transactions instead of aborting transactions later, in our analysis of the complexity of conservative schemes we have taken into account the cost of attempting to reschedule an operation that was previously made to wait. Further work still remains to be done on making the developed schemes fault-tolerant.

Acknowledgements

We would like to thank Nandit Soparkar for discussions that helped in the development of the paper.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

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

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 135–141, 1988.

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

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

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

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

[MRB+91] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The concurrency control problem in multidatabases: Characteristics and solutions. Technical Report TR-91-37, Department of Computer Science, University of Texas at Austin, 1991.

[MRKS91] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on Parallel and Distibuted Information Systems, Miami Beach, Florida*, 1991.

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

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the Fourth International Conference on Data Engineering, Los Angeles*, 1988.

[SKS91] N.R. Soparkar, H.F. Korth, and A. Silberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer*, December 1991.