Modular Synchronization in Distributed, Multiversion Databases: Version Control and Concurrency Control

Divyakant Agrawal and Soumitra Sengupta, Member, IEEE

Abstract—A version control mechanism is proposed that enhances the modularity and extensibility of multiversion concurrency control algorithms. We decouple the multiversion algorithms into two components: version control and concurrency control. This permits modular development of multiversion protocols, and simplifies the task of proving the correctness of these protocols. A set of procedures for version control is described that defines the interface to the version control component. We show that the same interface can be used by the database actions of both two-phase locking and time-stamp concurrency control protocols to access multiversion data. An interesting feature of our framework is that the execution of read-only transactions becomes completely independent of the underlying concurrency control implementation. Unlike other multiversion algorithms, read-only transactions in this scheme do not modify any version related information, and therefore, do not interfere with the execution of read-write transactions. Finally, the extension of the multiversion algorithms to a distributed environment becomes very simple.

Index Terms—Concurrency control, distributed databases, multiversion databases, read-only transactions.

I. INTRODUCTION

MULTIPLE versions of data are used in database systems to support transaction and system recovery. These multiple versions of data can also be exploited to improve the degree of concurrency that is achievable in the system. The higher concurrency is the result since out-of-order read requests can be serviced by reading appropriate older versions of data. Thus read-only transactions in most multiversion concurrency control schemes are executed almost unhindered. Specifically, the adverse effects of concurrent read-write transactions on read-only transactions are minimized. Unfortunately, in many multiversion concurrency control schemes, there is still a possibility of read-only transactions having undesirable effects on read-write transactions.

One of the observations that can be made about various multiversion concurrency control protocols is that each of them employs a different approach to integrate multiple versions of data with the desired concurrency control protocol. For example, the multiversion protocol with time-stamp ordering [17] is quite different from the multiversion protocol with two-phase locking [7]. This is so because the version control components of these algorithms are very closely tied to the chosen concurrency control protocols. In contrast, protocols for replicated data employ synchronization mechanisms which naturally divide into two components: the concurrency control component and the replica control component. The advantage of such subdivision is that it allows a modular development of new protocols and simplifies the task of proving correctness of these protocols. For example, a new concurrency control mechanism can be combined with the quorum protocol [11] for replica control very easily; the combined protocol can then be used for managing replicated data.

Unfortunately, no such subdivision exists for protocols that manage multiversion data. There are several advantages to the separation of the version control component from the concurrency control component. Conceptually, this separation allows modular development of multiversion protocols and simplifies the extension of these protocols to a distributed environment. Secondly, the task of proving correctness of such protocols is greatly simplified. Finally, there is improved extensibility in that more experimentations are possible in areas such as garbage collection algorithms and adaptive concurrency control schemes without introducing major modifications to the entire protocol.

In this paper, we propose a version control mechanism that can be integrated with any conflict-based concurrency control protocol, viz., two-phase locking [9], time-stamp ordering [17], or an optimistic concurrency control protocol [15]. One of the major advantages of the proposed version control mechanism is that read-only transactions do not have any concurrency control overhead, and cannot cause aborts of read-write transactions, as is the case in some other protocols [17]. The mechanism extends easily to distributed multiversion database environments. The version control mechanism guarantees global serializability of read-only transactions (unlike [8]), and the execution of read-only transactions is completely independent of the chosen concurrency control protocol.

The rest of this paper is organized as follows. In Section II, we briefly discuss other multiversion protocols and indicate their shortcomings. We present the formal model for correctness in multiversion databases in Section III. In Section IV, we present the version control mechanism, and demonstrate how it
can be integrated with the two-phase locking and time-stamp ordering protocols. Correctness of these protocols is argued in Section V. The version control mechanism is extended for a distributed environment in Section VI. Concluding remarks appear in Section VII.

II. MULTIVERSION ALGORITHMS

Numerous concurrency control algorithms have been proposed for multiversion databases [17, 19, 3, 7, 8], [21]. It is not our intention to propose yet another algorithm; instead, we propose a uniform methodology that can be used to implement these protocols. Multiversion time-stamp ordering was introduced by Reed [17]. The main advantage of this scheme is that read requests are never rejected. The algorithm can be viewed as an extension of the time-stamp ordering protocol [3]. However, there are several drawbacks associated with this algorithm as presented. First, read operations issued by read-only transactions in this protocol must be synchronized with the operations of read–write transactions, i.e., read operations may be blocked due to a pending write. Second, read-only transactions have a significant concurrency control overhead since they must update certain information associated with the versions of the objects. This may also result in a read-only transaction causing an abort of a read–write transaction. Finally, since read-only transactions update information in the database, distributed read-only transactions also require two-phase commit protocol for their atomic commitment. Thus, execution of read-only transactions in this protocol has a significant synchronization overhead.

Multiversion two-phase locking was originally proposed by Chan et al. [7]. This protocol makes a distinction between read-only and read–write transactions before the transactions begin execution. Read–write transactions are executed as in any other two-phase locking scheme, with some minor changes. Read-only transactions are handled differently in that additional information is associated with every read-only transaction. One is a start time-stamp which indicates the time when a transaction started, and other is a completed transaction list which is a list of all read–write transactions that have committed successfully until that time. One drawback of this scheme is the maintenance and usage of the completed transaction list. The execution of a read operation of a read-only transaction involves finding the largest version of an object which is smaller than the start time stamp of the transaction, and ensuring that the creator of this version appears in the copy of the completed transaction list of the transaction. This approach is cumbersome and complex to deal with.

The second drawback of the multiversion two-phase locking protocol [7] appears in its extension for distributed databases [8]. Although the distributed variant of the protocol guarantees a consistent view to a read only transaction, it does not guarantee global serializability of read-only transactions. The protocol also requires that a read only transaction must have a priori knowledge of the set of sites where it will perform its reads. This is necessary to construct a global completed transaction list from the copies of the local completed transaction lists at the respective sites before the read-only transaction begins its execution. Thus the complexity of the protocol increases when used in a distributed database environment.

Weihl proposed several protocols to implement read-only transactions and to manage multiversion databases [21]. We will discuss only one of them since others are primarily intended to integrate garbage collection algorithms efficiently. The protocol which uses time stamps and initiation [21] is similar to the multiversion two-phase locking algorithm [7]. In this protocol, a completed transaction list is not required; however, a read-only transaction has to perform synchronization actions with a concurrent read–write transaction to avoid inconsistent views. The synchronization is performed on time stamps associated with the objects, and in some cases, this may lead to a race condition where neither transaction may proceed with useful work.

We proposed a multiversion optimistic concurrency control protocol to primarily eliminate the validation overhead of read-only transactions [1], [2] in optimistic schemes. The mechanism presented in this paper is based on the version management scheme of the multiversion optimistic concurrency control protocol [2]. However, the concurrency control and version management issues are too closely interrelated in this protocol. As a result, it is difficult to determine if certain aspects of the protocol are necessary because of the optimistic protocol or due to the version management scheme.

The novel aspect of this paper is not that we present new multiversion algorithms, rather that it is possible to decouple the concurrency control issues from the version control issues. The proposed version control mechanism, to a certain degree, is based on all earlier multiversion algorithms. However, the elegance of this mechanism is that it allows simple and uniform integration with the standard concurrency control protocols, such as time-stamp ordering, two-phase locking, and optimistic schemes. To the best of our knowledge, no one has previously attempted to modularize the components of multiversion protocols as has been done for the replicated data management protocols [11], [20].

III. THE MODEL

A database consists of a set of objects, and users interact with the database system by invoking transaction programs. A transaction Ti is an ordered pair (Si, <i), where Si is the set of read and write operations in Ti that are executed atomically, and <i is an irreflexive partial order that represents the execution order of these operations. Read or write operations executed by Ti on an object x are denoted by rTi[x] and wTi[x] respectively.1 We assume that there is at most one rTi[x] and at most one wTi[x] in Si. Furthermore, if rTi[x] and wTi[x] are both in Si then rTi[x] <i wTi[x]. The execution of a transaction must appear atomic, i.e., a transaction Ti terminates with either a commit, ci, or an abort, ai, operation. The commit of a transaction results in all its changes being applied to the database; the abort results in the changes being discarded.

We refer to the application of read and write operations to database objects as operations. A read operation on two distinct objects corresponds to two distinct read operations.
The following definitions are borrowed from [6]. We denote the set of transactions that executed in a system as \( T = \{ T_1, \ldots, T_n \} \). The execution of transactions in \( T \) is modeled by a structure called history. A history, \( H \), over \( T \) is defined as a partial order \((\Sigma, <_H)\), where \( \Sigma \) is the set of all operations executed by transactions in \( T \), and \( <_H \) indicates the execution order of those operations.

### A. Single Version Data

We first describe the commonly accepted correctness criteria for single version databases. Let \( H \) be a history over a set of transactions \( T = \{ T_1, \ldots, T_n \} \). A transaction \( T_j \) reads \( x \) from another transaction \( T_i \) in \( H \) if:

1. \( w_i[x] <_H r_j[x] \).
2. \( \forall w_i[x] \in H \) such that \( w_i[x] <_H w_k[x] <_H r_j[x] \).

The final write of \( x \) in a history \( H \) is the operation \( w_i[x] \in H \), such that:

1. \( \forall w_j[x] \in H, w_j[x] <_H w_i[x] \).

Two histories, \( H \) and \( H' \), are equivalent if:

1. they are over the same set of transactions and have the same operations;
2. they have the same reads from relation for each object \( x \);
3. they have the same final writes for each object \( x \).

A serial history \( H_S \) is such that \( <_{H_S} \) is a total order and for every two transactions \( T_i \) and \( T_j \), either all operations of \( T_i \) precede all operations of \( T_j \), or vice-versa. A history is serializable if it is equivalent to a serial history. The serializable execution of transactions is a commonly accepted correctness criteria in database systems. However, the problem of determining if an arbitrary history is serializable is shown to be NP-complete [16]. Hence, concurrency control protocols for general purpose databases are based on the notion of conflict.

Two operations conflict if they both operate on the same object, and one of them is a write. A history \( H \) is conflict serializable if there exists some serial history \( H_S \) such that:

1. \( H \) and \( H_S \) are defined over the same set of transactions and have the same operations;
2. if \( o_1 \) and \( o_2 \) are two conflicting operations and \( o_1 <_H o_2 \), then \( o_1 <_{H_S} o_2 \).

It can be shown that \( H \) is equivalent to \( H_S \), and therefore, \( H \) is serializable. We can determine whether a history is serializable by analyzing a graph derived from the history called a serialization graph. The serialization graph for \( H \), denoted \( SG(H) \), is a directed graph whose nodes are the transactions in \( T \), and has an edge \( T_i \rightarrow T_j \) if one of \( T_i \)'s operations precedes and conflicts with one of \( T_j \)'s operations.

A history \( H \) is serializable if and only if \( SG(H) \) is acyclic [9],[ 3].

### B. Multiversion Data

We next consider a multiversion database in which each write operation on an object \( x \) produces a new version of \( x \). Thus for each object \( x \) in the database, there is a list of associated versions. A read operation on \( x \) is performed by returning the value of \( x \) from an appropriate version in the list. The existence of multiple versions is visible only to the scheduler implementing the protocol, and not to the user transactions which refer to the object as \( x \). The versions of \( x \) are denoted as \( x_1, x_2, \ldots \), where the subscripts are the monotonically increasing version numbers of each version.

The version number most often corresponds to the index or the transaction number of the transaction that wrote that version. We assume that if a transaction is aborted, all versions it created are discarded.

A multiversion (MV) history \( H \) over a set of transactions \( T \) represents the sequence of operations on the version of objects. Thus each \( w_i[x] \) in an MV history is mapped into \( w_i[x_i] \), and each \( r_j[x] \) into \( r_j[x_j] \), for some \( j \). A transaction \( T_j \) reads \( x \) from \( T_i \) in \( H \) if \( T_j \) reads a version of \( x \) produced by \( T_i \), i.e., \( r_j[x_i] \in H \). Note that the notion of final writes can be dropped from the definition of equivalence of multiversion history, since every write results in a new entity being created in the database [6].

Two MV histories over a set of transactions, \( T_i \), are equivalent if they have the same operations [6]. An MV history is one-version serializable if it is equivalent to a serial history over the same set of transactions executed over a single version database.

The serialization graph of an MV history \( H \) is a directed graph whose nodes represent transactions and whose edges are all \( T_i \rightarrow T_j \) such that one of \( T_i \)'s operations precedes and conflicts with one of \( T_j \)'s operations in \( H \). However, \( SG(H) \) by itself does not contain enough information to determine whether \( H \) is one-version serializable or not. To determine if an MV history is one-version serializable, a modified serialization graph is used. Given an MV history \( H \), a multiversion serialization graph (MVSG(\( H \))) is \( SG(H) \) with additional edges such that the following conditions hold.

1. For each object \( x \), MVSG(\( H \)) has a total order (denoted \( \leq \)) on all transactions that write \( x \).
2. For each object \( x \), if \( T_j \) reads \( x \) from \( T_i \) and if \( T_i \leq T_j \), then MVSG(\( H \)) has an edge from \( T_j \) to \( T_k \) (i.e., \( T_j \rightarrow T_k \)); otherwise, if \( T_k \leq T_i \), then MVSG(\( H \)) has an edge from \( T_k \) to \( T_i \) (i.e., \( T_k \rightarrow T_i \)).

The additional edges are called version order edges. An MV history \( H \) is one-version serializable if MVSG(\( H \)) is acyclic [4],[6].

### IV. THE VERSION CONTROL MECHANISM

In the following description of the version control mechanism, it is assumed that the execution of read–write transactions is synchronized by a concurrency control protocol that guarantees some serial order. Furthermore, a read–write transaction \( T \) is assigned a transaction number \( tn(T) \) which is unique and corresponds to the serial order. That is, if \( T_1 \) precedes \( T_2 \) in the serial order, then \( tn(T_1) < tn(T_2) \), and vice versa. It can be easily verified that any conflict-based concurrency control protocol can be changed to assign such numbers to the transactions. For example, in two-phase locking, read–write transactions can be assigned transaction numbers from a monotonically increasing counter when the transactions reach their lockpoint [16]. A transaction number
in time-stamp ordering simply corresponds to the logical time stamp of a transaction.

In this section we first describe the version control mechanism and also describe the execution of read only transactions using this mechanism. We next present multiversion algorithms in which version control is integrated with time-stamp ordering and with two-phase locking protocols. The multiversion algorithm with version control and optimistic concurrency control appears in [1], [2], and hence, is not presented in this paper. We conclude this section with a brief discussion of the proposed algorithms.

### A. Version Control

We now describe our version control mechanism integrated with an abstract concurrency control mechanism. First, we require that all transactions in the system be classified into the following two categories.

1. **Read-only transactions**: Transactions which do not modify the state of the database, and therefore, do not execute any write operations.
2. **Read-write transactions**: Transactions which update the state of the database, and therefore, execute at least one write operation.

If a transaction's class cannot be determined a priori, it is classified as a read–write transaction by default. Since we assume that all read–write transactions are serialized by the underlying concurrency control protocol, a transaction with an unknown category will be serialized with respect to read–write transactions.

The read–write transactions execute in the multiversion environment in the same way as in a single version environment. That is, the concurrency control related synchronization for the read-write transactions is performed as if a single version of an object exists in the database. The read operation is carried out by reading the most recent version of the object, and a write operation creates a new version of the object.

The version number of the version of the object written by a read–write transaction is its transaction number. In order to assign these numbers to the transactions, the version control mechanism maintains a monotonically increasing counter called the transaction number counter, \( \text{tn} \).

Let us consider the read-only transactions next. If we make the versions of data objects visible to a read-only transaction in such a way that no smaller version can be created by any active or future transactions, we can easily serialize the read-only transactions in the system. This is accomplished by first choosing a value of \( \text{tn} \) such that all transactions with transaction numbers smaller than the chosen value have completed. This value is used to assign a number to the read-only transactions when they begin execution, and is called the start number of a transaction, \( sn(T) \). Later, when \( T \) reads an object \( x \), it chooses the largest version of \( x \) that is smaller than \( sn(T) \). It can be informally argued that \( T \) is serialized according to its \( sn(T) \), i.e., it succeeds all completed read–write transactions and precedes all active and future read–write transactions.

The main problem in executing read-only transactions is how to choose an appropriate value of the start number that will guarantee the property mentioned previously. This is precisely the source of complication in multiversion two-phase locking as proposed in [7] and [21]. A transaction \( T \) is assigned \( tn(T) \) at the beginning in multiversion time-stamp ordering and it is assigned \( tn(T) \) at the lock-point in multiversion two-phase locking. Since in both schemes \( T \) remains active after the assignment, the current value of \( \text{tn} \) cannot be used to assign start numbers to the read-only transactions because its writes may not be complete. In our scheme we employ another monotonically increasing counter called the visible transaction number counter \( \text{vtnc} \). Intuitively, the value of \( \text{vtnc} \) controls the visibility of the versions of data objects to the read-only transactions. Hence, \( \text{vtnc} \) serves the purpose of assigning start numbers to the read-only transactions. Unlike \( \text{tn} \), which is incremented when a transaction is assigned a transaction number, \( \text{vtnc} \) may be incremented only when a transaction completes. It will be left unchanged, however, if, at the time a transaction \( T \) completes, there is another transaction \( T' \) that is still active and \( tn(T') < tn(T) \). This is possible since the transaction number order need not necessarily correspond to the order in which transactions complete their execution. Thus \( \text{vtnc} \) is incremented in such a way that the versions of data objects are made visible according to the serialization order of transactions in the system. Hence, we can state the following properties for the two counters in our scheme:

1. **Transaction Ordering Property**: The value of \( \text{tn} \) at all times is the largest number such that all transactions \( T \), which either are active and unassigned or will arrive later, will have \( tn(T) \geq \text{tn} \).

2. **Transaction Visibility Property**: The value of \( \text{vtnc} \) at all times is the largest number such that all transactions \( T \) with \( tn(T) \leq \text{vtnc} \) have completed.

Additionally, the values of the two counters must be such that \( \text{vtnc} < \text{tn} \) at all times.

The interface to the version control component is illustrated in the module `VersionControl` in Fig. 1. It has four entry procedures: `VStart()`, `VRegister(T, status)`, `VDiscard(T)`, and `VComplete(T)`. Also, this module maintains three data structures related to version control: \( \text{tn} \), \( \text{vtnc} \), and \( \text{VCQueue} \). Since the procedures access and update these shared structures, they must be executed atomically. In other words, if a procedure is executed by a transaction, no other procedure may be executed concurrently by another transaction. The critical section implementation to ensure this characteristic is assumed as part of the module declaration and implementation.

\( \text{VCQueue} \) is an ordered list of all transactions that have been assigned a transaction number (and therefore, have a fixed position in the serial order) and are still active in the system or are waiting for a transaction with a smaller transaction number to complete. This queue is used to make the versions created by the read-write transactions visible in the order of their serialization. There are four operations defined on \( \text{VCQueue} \).

The **Insert** operation inserts a transaction entry in \( \text{VCQueue} \) according to its transaction number while maintaining the order of entries in \( \text{VCQueue} \). The **Discard** operation deletes
**B. Read-Only Transactions**

The execution of read-only transactions is shown in Fig. 2. The left-hand column shows the action of a read-only transaction, and the right-hand column illustrates the resulting execution of the same transaction. The read-only transactions in our scheme are independent of the underlying concurrency control protocol. These transactions do not interact with the concurrency control module at all, and make only one call to the version control module. Therefore, there is almost negligible overhead associated with read-only transactions.

A read only transaction, \( T \), begins its execution by obtaining its start number, \( \text{sn}(T) \), from the version control module. Note, that \( \text{sn}(T) \) for read-only transactions is also \( \text{tn}(T) \). A read request of \( T \) for an object \( x \) is never blocked and results in the transaction reading a version of \( x \) with the largest version number less that or equal to \( \text{sn}(T) \). Barring the unavailability of an appropriate version to read due to garbage collection of old versions, a read request of \( T \) is never rejected.

**C. Version Control with Time-Stamp Ordering**

Since the serial order of transactions in time-stamp ordering is determined \textit{a priori}, read-write transactions are assigned a transaction number before they begin execution. Fig. 3 illustrates the execution of read-write transactions in a time-stamp ordering protocol integrated with version control. The procedure \( \text{VCregister}(T, \text{"active"}) \) executed by a read-write transaction, \( T \), serves the purpose of assigning a number to \( T \) and registering it for version control purposes. The rationale behind registering \( T \) in the \( \text{VCQueue} \), as soon as 'T's serial order is determined, is to ensure that we do not make updates of latter transactions visible before \( T \) completes. If this is not enforced, it will result in nonserializable execution of read-only transactions. Note that in time-stamp ordering protocol \( \text{sn}(T) \) is the same as \( \text{tn}(T) \).

Recall that in time-stamp ordering protocols [17], [3], each version of an object \( x \) has a unique write time stamp, \( w \cdot \text{ts}(x) \), which records the transaction number of the transaction that created this version of \( x \). The most recent version of \( x \) additionally has a read time stamp, \( r \cdot \text{ts}(x) \), which records the time stamp of the youngest transaction that read the most recent version of \( x \).
A write request from $T$ for $x$ is granted only if $tn(T)$ is greater than or equal to the read and the write time stamps of the most recent version of $x$. This conflict check is required in order to serialize transactions in their time-stamp order. Transactions whose write requests are not granted are aborted. Once a write request is granted, it is considered pending until the writer commits. If a read or write request is made for an object by a transaction, and there exists a pending write request for the object by an older transaction, the read or write request is blocked until the pending write is no longer pending, i.e., until the older transaction either commits or aborts.

Read requests are never rejected, though they may sometimes be blocked due to pending write requests. A read request from a transaction $T$ for an object $x$ is granted by allowing the transaction to read a version of $x$ with the largest $w-ts(x)$ such that $tn(T) \geq w-ts(x)$. Note that, although $T$ must have started more recently than the writer of this version of $x$, the writer may still be executing. This is the case that requires that a read request be blocked for the completion of the pending write request.

At the time of termination of $T$, we check $VCQueue$ to see if its updates are subject to delayed visibility on account of older transactions that are still active. If there are no such transactions, $vtnc$ is set to $tn(T)$. Otherwise the increment of $vtnc$ is delayed until the future time when these older transactions complete their execution. Also, if $T$ is aborted for any reason after it has been in $VCQueue$, its entry must be discarded from $VCQueue$. This ensures that the visibility is delayed due to active and un aborted transactions only.

**D. Version Control with Two-Phase Locking**

In a two-phase locking protocol, the serial order of read-write transactions corresponds to their lock-points. A lock-point of a transaction is a point in time between the last lock acquired and the first lock released by a transaction. Thus while the transaction is executing its read and write operations, i.e., acquiring additional locks, its serial order is uncertain. Therefore, a read-write transaction in this scheme is not registered with the version control module until it completes its execution phase. We assume that the execution phase of a transaction, $T$, is complete when it invokes the action $end(T)$. The execution of a read-write transaction in a two-phase locking protocol integrated with version control is illustrated in Fig. 4.

A read-write transaction, $T$, in two-phase locking scheme always reads the latest version of objects. Hence, for sake of uniformity, $sn(T)$ is chosen as infinity. A read request from $T$ for $x$ results in obtaining a read lock on $x$. If the lock is not available, $T$ is delayed. Otherwise, $T$ reads the largest version of $x$ in the database. Since in two-phase locking a lock may be released only after a transaction has reached its lock-point, $T$'s acquiring the read lock guarantees that there are no write locks on $x$, and also that any transaction $T'$ that wrote $x$ must have had a lock-point smaller than $T$, and hence, must precede $T$ in the serial order. Similarly, since $T$ will release its lock on $x$ only after it has reached its lock-point, any other transaction $T'$ that intends to write $x$ will have to wait for the lock release, and thus will have a lock-point larger that $T$, and hence, will succeed $T$ in the serial order. It is thus guaranteed that after $T$ acquires the read lock on $x$, the version of $x$ that $T$ reads is the latest version.

A similar argument shows that $T$ is the only transaction that writes the noncommitted version of $y$ after it acquires a write lock on $y$. The difference from the time-stamp ordering protocol, however, is that $T$ does not have an assigned $tn(T)$ as yet. However, this is not a problem because this protocol will not allow any other transaction to read the version of $y$ created by $T$ until it releases its lock on $y$. If $T$ releases its lock on $y$, it must have gone past its lock-point and must have been assigned $tn(T)$. Thus once again, the version control mechanism requires that $T$ be registered as soon as its serial order is determined. At the end of execution, $T$ is registered in the $VCQueue$ and thus is assigned $tn(T)$ in the order of its lock-point, i.e., in the order of its serialization. $T$ can then complete its database updates using $tn(T)$ as the version number.

This scheme delineates read-only and read-write transactions completely. Since a read-only transaction execution is independent of the concurrency control, it is unaffected by the concurrent read-write transactions (unlike [7] and [21]), and the algorithms are considerably simpler. The version control mechanism is not affected by deadlocks that may arise in the system since the transactions that interact with the version control have gone past their lock-point. Since such transactions cannot have any pending lock requests, they cannot be involved in any deadlock cycle.

**E. Discussion**

Read-only transactions in a multiversion algorithm with the version control mechanism do not interact with the concurrency control component; therefore, the multiversion algorithm with version control does not have any synchronization overhead for read-only transactions. Furthermore, unlike multiversion time-stamp ordering, the version control mechanism guarantees that a read-only transaction cannot delay or abort read-write transactions. Also, the execution of
read-only transactions is relatively simple when compared to that in the multiversion two-phase locking protocol. Finally, multiversion algorithms with the version control mechanism described in this paper do not need an extended negotiation phase for a transaction’s serialization order as is the case in the multiversion protocol presented in [21].

In order to achieve the advantages mentioned previously, the version control mechanism is deficient in one aspect of transaction execution. The deficiency is that the read-only transactions suffer from delayed visibility due to the lag between the two counters, and there are several techniques to rectify this problem. First, delayed visibility may violate the temporal relationship between transactions. For example, a read-only transaction, \( R \), executed immediately after a read-write transaction, \( T \), may not see the results of \( T \). If unacceptably, this problem can be rectified by ensuring that \( R \) be executed with a value of \( sn(R) \) which is at least as large as \( tn(T) \). The second and more serious shortcoming of delayed visibility is that read-only transactions do not observe the most recent state of the database. Although this may be acceptable to most read-only transactions, some applications may not be willing to sacrifice currency at the expense of concurrency [10]. Such transactions can be dealt with by executing them as pseudo-read-write transactions.

No description of a multiversion algorithm is complete unless some attention is given to the process of garbage collection of old and unnecessary versions of data. In our scheme, a garbage collection algorithm, which keeps the information about read-only transactions, can be integrated easily. The only restriction the version control mechanism imposes on the garbage collection scheme is that it must not discard any version of objects as young as or younger than \( vtac \) or \( sn(T_{oldest}) \), where \( T_{oldest} \) is the oldest transaction in the system. This separation again helps since the concurrency control component is not overloaded with auxiliary functions. Also, the garbage collection scheme does not interact with the read-write transactions and the concurrency control component does not interact with the read-only transactions. We feel that this separation is quite elegant and desirable.

V. Correctness

In this section we demonstrate that the two multiversion algorithms developed in the previous section guarantee serializability of all transactions.

A. Proof of Version Control with Time-Stamp Ordering

The following lemmas state certain properties of this protocol. We will use these properties to prove that the protocol is one-version serializable. The first lemma indicates that the read-write transactions in this protocol are assigned unique transaction numbers. Note that the lemma holds only for read-write transactions; unlike the multiversion time-stamp ordering protocol, several read-only transactions in this scheme may be assigned the same transaction number. However, this fact does not affect the proof of serializability.

**Lemma 1:** For each read-write transaction \( T_i \) there is a unique transaction number \( tn(T_i) \).

**Proof:** Immediate from the assumption that transaction number assignment is unique in the time-stamp ordering protocol.

The next lemma states that a transaction reads versions of objects that were created by its predecessors in the serial order.

**Lemma 2:** For every \( r_k[x_j] \in H, \ w_j[x_j] < r_k[x_j] \) and \( tn(T_j) \leq tn(T_k) \).

**Proof:** Consider the two cases: one is when the read operation \( r_k[x_j] \) corresponds to a read-write transaction \( T_k \) and the other is when \( r_k[x_j] \) corresponds to a read-only transaction.

If \( T_k \) is a read-write transaction, the time-stamp concurrency control protocol (through the version control action VCRegister) assigns \( tn(T_k) \) before \( T_k \) executes any of its operation. Furthermore, \( r_k[x] \) is executed by choosing a largest version \( x_j \) such that \( tn(T_j) < tn(T_k) \).

If \( T_k \) is a read-only transaction, the version control protocol (through the action VCStart) assigns \( sn(T_k) \) before \( T_k \) executes any of its operation. Since \( T_k \) is a read-only transaction, \( tn(T_k) = sn(T_k) \). The execution of \( r_k[x] \) will return \( x_j \) such that \( tn(T_j) \leq sn(T_k) \). Hence, \( tn(T_j) \leq tn(T_k) \).

The following lemma states that when a transaction, \( T \), reads an object \( x \), it reads a version of \( x \) which is the largest version smaller than \( tn(T) \). In addition, if another transaction later attempts to write \( x \) with a transaction number smaller than \( tn(T) \) and larger than the version of \( x \) read by \( T \), the write will be rejected and the transaction will be aborted.

**Lemma 3:** For every \( r_k[x_j] \in H \) and \( w_i[x_i] \in H, i \neq j \), one of the following conditions must hold:

1) \( tn(T_i) < tn(T_j) \), or
2) \( tn(T_k) < tn(T_i) \), or
3) \( i = k \) and \( r_k[x_j] < w_i[x_i] \).

**Proof:** Consider the two cases when \( T_k \) is a read-write transaction and when \( T_k \) is a read-only transaction.

**Case 1:** \( T_k \) is a read-write transaction. Let us assume \( i \neq k \). From Lemma 2, \( tn(T_j) < tn(T_k) \). Furthermore, the time-stamp concurrency control protocol would reject an operation \( w_i[x_i] \) if \( tn(T_i) < tn(T_j) \). Hence, the only possibilities are either \( tn(T_i) < tn(T_j) \) or \( tn(T_k) < tn(T_i) \). The case \( i = k \) holds due to the restriction on the transactions (see Section III).

**Case 2:** \( T_k \) is a read-only transaction. From Lemma 2, \( tn(T_j) \leq tn(T_k) \). Since \( T_j \) is a read-only transaction, \( i \neq k \). Consider the case when \( tn(T_j) = tn(T_k) \). From the uniqueness of transaction numbers of read-write transactions, it follows that either \( tn(T_i) < tn(T_j) \) or \( tn(T_k) < tn(T_i) \).

Consider the case when \( tn(T_j) < tn(T_k) \). The transaction ordering property and transaction visibility property together guarantee that it is not possible to have \( w_i[x_i] \) such that \( tn(T_i) < tn(T_j) \leq tn(T_k) \). Therefore, there are only two possibilities: either \( tn(T_j) < tn(T_i) \) or \( tn(T_k) < tn(T_i) \).

By using the above lemmas as formal specifications of the protocol, the following theorem demonstrates that the protocol guarantees one-version serializability. The theorem is an extension of the theorem for the multiversion time-stamp ordering protocol [6].
Theorem 1: Version control with time-stamp ordering guarantees serializable execution of transactions.

Proof: We define the version order as \( \ll_x \) for an object \( x \) as the total order on the transaction numbers, of the transactions creating versions of \( x \), i.e., \( x_i \ll_x x_j \) if and only if \( t_n(T_i) < t_n(T_j) \).

Let \( H \) be a history produced by the version control with time-stamp ordering protocol. We will prove that \( MVSG(H) \) is acyclic by showing that for each edge \( T_i \to T_j \) in \( MVSG(H) \), \( t_n(T_i) < t_n(T_j) \).

Recall that \( MVSG(H) \) includes edges in \( SG(H) \) and additional version order edges. Consider the edges in \( SG(H) \). Each edge \( T_i \to T_j \) in \( SG(H) \) is due to a reads-from relation, i.e., for some \( x \), \( T_j \) reads \( x \) from \( T_i \). From Lemma 2 and Lemma 1, \( t_n(T_i) \leq t_n(T_j) \). For any edge \( T_i \to T_j \) involved in a cycle, \( T_j \) has to be a read-write transaction. Hence, \( t_n(T_i) < t_n(T_j) \).

Next consider the version order edges in \( MVSG(H) \). Let \( r_k[x_j] \) and \( w_k[x_i] \) be in \( H \) where \( i, j, \) and \( k \) are distinct. Consider the following cases:

1) \( x_i \ll_x x_j \), which implies \( T_i \to T_j \) is in \( MVSG(H) \);
2) \( x_j \ll_x x_i \), which implies \( T_k \to T_j \) is in \( MVSG(H) \).

In case 1, from the definition of version order, \( t_n(T_i) < t_n(T_j) \). In case 2, from Lemma 3, \( t_n(T_j) < t_n(T_k) \) or \( t_n(T_k) < t_n(T_i) \). Since \( x_j \ll_x x_i \), \( t_n(T_i) < t_n(T_j) \) is not possible. Hence, \( t_n(T_k) < t_n(T_i) \).

If \( MVSG(H) \) has a cycle, it violates the total order of the transaction numbers of transactions involved in that cycle. Thus by the application of the serializability theorem for multiversion data, every history \( H \) produced by the version control with time-stamp ordering protocol is one-version serializable.

B. Proof of Version Control with Two-Phase Locking

The proof of this protocol is very similar to the earlier protocol. The following lemmas state the properties of this protocol, which are needed to prove the serializable execution of transactions in this protocol. An interesting observation about these lemmas is that the analysis needs to be done only for read-write transactions. Since read-only transactions are not influenced by the concurrency control protocol, the properties proved in Section V-A still hold.

Lemma 4: For each read-write transaction \( T \), there is a unique transaction number \( t_n(T) \).

Proof: Immediate from the assumption that transaction number assignment is unique in two-phase locking protocol.

As before, the next lemma states that a transaction reads versions of objects that were created by its predecessors in the serial order.

Lemma 5: For every \( r_k[x_j] \in H \), \( w_j[x_j] < r_k[x_j] \) and \( t_n(T_j) \leq t_n(T_k) \).

Proof: The case when \( T_k \) is a read-only transaction has already been analyzed in Lemma 2. Thus we only consider the case when \( T_k \) is a read-write transaction.

If \( T_k \) is a read-write transaction, the two-phase locking concurrency control protocol (through the version control action VCRegister) assigns \( t_n(T_k) \) after \( T_k \) has obtained all its locks. Since \( T_k \) reads \( x \), it has a read lock on \( x \). Furthermore, since \( T_k \) reads the version \( x_j \), it implies that a transaction \( T_j \) acquired a write lock on \( x \) and released it prior to when \( T_k \) acquired the read lock. This means that the lock-point of \( T_j \) precedes that of \( T_k \). Since the version control mechanism assigns transaction numbers in the lock-point order, it must be the case that \( t_n(T_j) < t_n(T_k) \).

Hence \( t_n(T_j) \leq t_n(T_k) \), for any transaction \( T_k \).

We now prove that transactions read most recent version of objects in this protocol.

Lemma 6: For every \( r_k[x_j] \) and \( w_i[x_i] \in H \), \( i \neq j \), one of the following conditions must hold:
1) \( t_n(T_i) < t_n(T_j) \), or
2) \( t_n(T_k) < t_n(T_i) \), or
3) \( i = k \) and \( r_k[x_j] < w_i[x_i] \).

Proof: Consider the case when \( T_k \) is a read-write transaction. Let us assume \( i \neq k \). From Lemma 5, \( t_n(T_j) < t_n(T_k) \). If \( t_n(T_j) < t_n(T_i) < t_n(T_k) \) and there exists \( w_i[x_i] \), \( T_i \) must have released the write lock on \( x \) after creating \( x_i \), but before \( T_k \) was granted a read lock on \( x \). In this case \( T_k \) should have read \( x_i \) (and not \( x_i \)), which is a contradiction. Therefore, since \( r_k[x_j] \in H \), either \( t_n(T_i) < t_n(T_j) \) or \( t_n(T_k) < t_n(T_i) \).

The case \( i = k \) holds due to the restriction on the transactions (see Section III).

By using the above lemmas as formal specifications of the protocol, the following theorem demonstrates that the protocol guarantees one-version serializability [5], [6].

Theorem 2: Version control with two-phase locking guarantees serializable execution of transactions.

Proof: Identical to Theorem 1.

VI. DISTRIBUTED VERSION CONTROL MECHANISM

We consider a model of a distributed database which consists of a set of sites stored at several sites in a computer network. It is assumed that objects are not replicated. As was the case in the centralized multiversion database, each object at a site in the distributed multiversion database is stored as a monotonically increasing sequence of versions. Each site maintains its own version control module and counters. Transactions that access objects at multiple sites are termed as global transactions. The site at which a global transaction is initiated is called its coordinator, \( S_c \). Other sites which participate in the execution of the global transaction are called its cohorts, \( S_i \). A two-phase commit protocol [12] is employed to guarantee the atomicity of global transactions in the presence of site failures.

In this section, we first present the extensions necessary to the version control module for executing global transactions. We then describe the execution of global read-only transactions and global read-write transactions. The algorithms presented in this section underscore the simplicity of distributed version control mechanism. We conclude this section with a discussion of some of the optimizations that may improve the performance in the system.
MODULE VersionControl
PERSISTENT DATA D Cm, wtnci, COUNT, VCQueue, QUEUE;
PROCEDURE VCcreate(T : Transaction, Coord : String) BEGIN;
  COUNT := COUNT + 1;
  wtnci := wtnci + 1;
  VCQueue := VCQueue + (wtnci, Coord);
  Coord := Coord + "\n";
END;
PROCEDURE VCregister(T : TransactionNum, Coord : String) BOOLEAN;
  IF (vtnc) = NULL THEN
    vtnc := vtnc + 1;
    RETURN(TRUE);
  ELSE
    vtnc := vtnc + 1;
    RETURN(FALSE);
  END;
END;
PROCEDURE VCregister(T : TransactionNum) BOOLEAN;
  VCregister(T, Coord);
END;
PROCEDURE VCcomplete(T : TransactionNum, Coord : String) BOOLEAN;
  IF (vtnc) = NULL THEN
    vtnc := vtnc + 1;
    RETURN(TRUE);
  ELSE
    vtnc := vtnc + 1;
    RETURN(FALSE);
  END;
END;
PROCEDURE VCQueue; RETURN(QUEUE);
END;
PROCEDURE VCComplete(T : TransactionNum);
  Begin(T) := Begin(T) - vtnc;
  End(T) := End(T) + vtnc;
END;
PROCEDURE VCQueue; RETURN(QUEUE);
END;

A. Distributed Version Control

The distributed algorithm guarantees serializability of a global transaction by requiring that it be assigned and serialized with the same transaction number at all cohorts. This ensures that if there are two global transactions involving common cohorts, then at each site the serial order of the two transactions will be the same. Similarly, global read-only transactions are guaranteed a consistent global view of the database by requiring that they use the same start number to access data at all cohorts.

The centralized version control mechanism developed in Section IV extends naturally to a distributed environment. Each database site maintains the concurrency control and the version control modules. The data-structures necessary to implement these modules at a site are identified by the subscript i (e.g., wtnci is the visible transaction number counter at Si). The version control module is slightly modified to accommodate execution of global transactions.

The distributed version control module employed at a site Si is illustrated in Fig. 5. There are two major changes in this module. Since a global transaction is assigned a transaction number at its coordinator, it registers at its cohorts with the same transaction number. The transaction number of T is made globally unique by using the transaction identifier as the fractional part of tsn(T). Also, T cannot be registered at its cohort Si if the transaction visibility property is violated at Si. The procedure VCRegister() has been modified to register transactions in VCQueue, with the assigned transaction number. The other change is related to the data-structure VCQueuei, which now includes entries of type "waiting" of global read-only transactions. The procedure VCComplete() has been updated to incorporate this change.

B. Global Read-Only Transactions

The execution of a global read-only transaction is illustrated in Fig. 6. Note that we use x@Si to denote that object x at site Si is being accessed and procedure@Si to denote that procedure at site Si is being invoked. In order to provide a consistent view to a global read-only transaction T, one approach is to collect values of vtnci from every site where T will access objects, and assign the minimum value of all vtnci as the start number of T. However, because we make no assumptions about the set of sites where T will access the objects a priori, this is not a viable approach. Instead, T begins its execution by obtaining its start number, sn(T), from the version control module at its coordinator, and uses the locally assigned value of sn(T) to access objects at every site in the system.

When T accesses an object x at site Si, and if sn(T) ≤ vtnci, the access of T at Si is permitted by reading a version of x with the largest version number less than or equal to sn(T). Since the counters at various sites are not strictly synchronized, it is possible that T attempts to access x at Si where sn(T) > vtnci. In this case, the execution of T at Si must be delayed until vtnci becomes at least as large as sn(T), i.e., until all transactions at Si, with transaction numbers smaller than or equal to sn(T) have completed. This ensures that the transaction visibility property is not violated at Si, and is accomplished by registering a "waiting" entry in

The distributed algorithms ensure that each site individually satisfies the transaction visibility property. The transaction ordering property is no longer guaranteed for all transactions registering at a site. However, the consequence of this property, that of unique transaction numbers, is still guaranteed by using the transaction identifiers as the fractional part of the transaction numbers. Recall that the correctness proof depended on the fact that the transaction numbers be unique; the transaction ordering property, however, was not necessary for the correctness. Finally, in the distributed version control mechanism, since each read-only transaction uses a single value of a start number to access data and each read-write transaction uses a single value as its transaction number, the correctness proof of Section V still holds. Note that the correctness of the protocol was dependent on the uniqueness of transaction numbers and the maintenance of transaction visibility property.
AGRAWAL AND SENGUPTA: MODULAR SYNCHRONIZATION IN DISTRIBUTED MULTlVERSION DATABASES 135

Fig. 7. Execution of global read-write transactions in the time-stamp ordering protocol.

VCQueue, with \(tn(T) = sn(T)\). When \(vtnc\), becomes equal to \(sn(T)\) at \(S_i\), the entry of \(T\) is deleted and \(T\) is permitted to access the appropriate version of \(x\) at \(S_i\) (see procedure \(VC\text{complete}\) in Fig. 5).

Note that the procedure \(VC\text{register}\) returns FALSE only if \(tn(T)\) is less than or equal to \(vtnc\), which for a read-only transaction implies that it is allowed to access the object. The action execution for the read operation, therefore, does not need to be in a critical section even though the same version control data \((vtnc)\) is accessed more than once. The correctness requirement remains the same as in the single site case, that is, only the version control procedures need to be executed atomically.

The correct execution of global read-only transactions is guaranteed since they use a single value of start numbers to access all objects. This rule accompanied with the transaction visibility property enforced at every site ensures that a global read-only transaction will observe a consistent snapshot of the database. Also, the execution of such transactions is independent of the underlying concurrency control mechanism. Furthermore, deadlock is not possible since the waiting is unidirectional, i.e., read-only transactions are blocked for the completion of read-write transactions and not vice-versa.

C. Distributed Version Control with Time-Stamp Ordering

The extension of the version control mechanism to implement the execution of global read write transaction is done by ensuring that a single transaction number is associated with every read write transaction. That is, global read-write transactions are assigned a transaction number at their coordinator, and this number is used at every site for their execution. The execution of global read-write transactions is illustrated in Figs. 7 and 8. Fig. 7 depicts the invocation and corresponding execution of the actions of such transactions. The two-phase commit protocol used is shown in Fig. 8, which is invoked at the time when a transaction terminates.

Since the serial order of transactions in time-stamp ordering is determined a priori, a global transaction \(T\) is assigned a transaction number, \(tn(T)\), at its coordinator \(S_c\) before it begins its execution. The procedure \(VC\text{register}(T; "active")\) is executed by \(T\) to obtain \(tn(T)\), and it ensures that \(vtnc\) will not exceed \(tn(T)\) until \(T\) completes its execution at \(S_c\). This guarantees that the transaction visibility property will be maintained at the coordinator. In order to maintain the transaction visibility property at the cohorts of \(T\), it must also be registered with the version control modules at the cohorts before it makes an access at these sites. This is accomplished by registering \(T\) at site \(S_i\) with \(tn(T)\) when it makes the first access at \(S_i\). The condition for the first access is tested by checking if the subtransaction of \(T\) is known at \(S_i\). If not so, and it is found that \(vtnc\) is larger than \(tn(T)\), \(T\) is aborted (so that the transaction visibility property is not violated at \(S_i\)). Otherwise, \(T\) is registered at \(S_i\) and it can read and write objects at \(S_i\) in a manner identical to the read and write operations in the centralized case (Section IV-C).

When the global read-write transaction \(T\) terminates, a two-phase commit protocol illustrated in Fig. 8 is initiated on its behalf. During the first phase of the protocol, the coordinator of \(T\) requests all its cohorts to make the updates of \(T\) recoverable at their respective sites. After collecting the votes from all cohorts, the coordinator enters the second phase of the protocol. In this phase, if the coordinator discovers that some cohort has sent a negative vote (perhaps due to a local failure), it aborts \(T\) and sends its decision to all cohorts. Otherwise, \(T\) is committed at \(S_c\) and it can read and write objects at \(S_i\) in a manner identical to the read and write operations in the centralized case (Section IV-C).
Fig. 9. Execution of global read-write transactions in the two-phase locking protocol.

<table>
<thead>
<tr>
<th>Coordination $S_c$</th>
<th>Cohorts $S_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PHASE I</strong></td>
<td></td>
</tr>
<tr>
<td>VCRegister($T$, &quot;action&quot;)@$S_c$;</td>
<td></td>
</tr>
<tr>
<td>send(&quot;prepare&quot;) to all cohorts;</td>
<td></td>
</tr>
<tr>
<td>IF VCRegister($T$, &quot;action&quot;)@$S_i$, THEN</td>
<td></td>
</tr>
<tr>
<td>receive(&quot;prepare&quot;) from the coordinator;</td>
<td></td>
</tr>
<tr>
<td>Make updates recoverable; vote ~ YES;</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>Abort($T$); vote ~ NO;</td>
<td></td>
</tr>
<tr>
<td>END;</td>
<td></td>
</tr>
<tr>
<td>send(vote) to the coordinator;</td>
<td></td>
</tr>
<tr>
<td><strong>PHASE II</strong></td>
<td></td>
</tr>
<tr>
<td>As in Figure 8</td>
<td>As in Figure 8</td>
</tr>
</tbody>
</table>

Fig. 10. Two-phase commit protocol with two-phase locking.

D. Distributed Version Control with Two-Phase Locking

The execution of global read-write transactions in this protocol is illustrated in Figs. 9 and 10. Recall that the read-write transactions interact with the version control mechanism only at the end of their execution phase. As a consequence of this property, the invocation and the corresponding execution of the actions of such transactions, shown in Fig. 9, is identical to that described in the centralized environment, the only distinction being that the lock acquisition and the read or write operation on an object is performed at the site where the object resides.

A global transaction, $T$, obtains a transaction number $tn(T)$ from its coordinator (by invoking VCRegister at $S_c$) before initiating the two-phase commit protocol, which is illustrated in Fig. 10. During the first phase of the commit protocol, all cohorts try to register $T$ with $tn(T)$ at their respective sites. If the transaction visibility property is violated at any cohort $S_i$, i.e., $vtnc_i > tn(T)$, $T$ is aborted. On the other hand, if $T$ can be registered at all cohorts with the assigned transaction number, it enters the second phase of the protocol. This phase is very similar to the second phase of the two-phase commit protocol of time-stamp ordering with distributed version control. The similarity of the two-phase commit protocol is primarily due to the modularity of the version control mechanism.

E. Discussion

In the distributed version control mechanism described previously, global read-only transactions are not completely independent of the concurrent read-write transactions. In particular, a global read-only transaction $R$ may have to wait at a cohort $S_i$ for $vtnc_i$ to become at least as large as $sn(R)$. The increment of $vtnc_i$ is dependent upon read-write transactions. An obvious improvement is that if there are no active read-write transactions $T$ with $tn(T)$ between $vtnc_i$ and $sn(R)$, then $vtnc_i$ (and $tnco_i$, if necessary) can be set to $sn(R)$ and $R$ can be allowed to execute at $S_i$. If the waiting of $R$ is to be prevented completely, it is necessary to know the cohorts of $R$ in advance so that $sn(R)$ can be computed as the minimum value of $vtnc_i$ at all cohorts $S_i$.

Another issue of the distributed algorithm is related to the fact that in a distributed, autonomous environment, it is not possible to have a perfect knowledge of all counter values at all sites. This results in the aborts of global read-write transactions when their registration at a site violates the transaction visibility property. By employing simple modifications, this problem can be reduced in the time-stamp ordering protocol and can be eliminated in the two-phase locking protocol.

In time-stamp ordering, if the logical clocks or the counters are not kept in close synchrony, there is always a danger of excessive aborts of global transactions on account of out-of-order read or write requests. The only solution to minimize this problem is to keep various clocks or counters in approximate synchrony. A similar approach will result in reducing the transaction aborts on account of the violation of the transaction visibility property.

In two-phase locking, the distributed version control mechanism can be modified to use the transaction number assignment of a global transaction $T$ at its coordinator only as a “hint.” If all cohorts are able to register $T$ with $tn(T)$, the two-phase commit protocol remains the same as before. On the other hand, if some cohort $S_i$ is unable to register $T$ with $tn(T)$, it registers $T$ with $tn'(T)$ such that the transaction visibility property at $S_i$ holds (for example, $tn'(T)$ may be assigned the incremented value of $tnco_i$). Note that $tn'(T)$ must necessarily be larger than $tn(T)$. This introduces an intermediate phase in the commit protocol, during which the coordinator collects all the new transaction number assignments of $T$, computes the maximum, and uses that to register $T$ at all cohorts. This reassignment is possible in two-phase locking because the transaction numbers are negotiable in the interval $(\text{Maximum}(vtnc_i), \infty)$. This is in contrast to the time-stamp ordering protocol where the transaction numbers are predetermined and fixed.

In the distributed scheme, the extension for garbage collecting the old versions of data is simple and quite straightforward. The garbage collection module at a site must keep track of local as well as global read-only transactions that executed at this site. As before, the module does not interact with the read-write transactions. An alternative implementation could be the one in which the garbage collection module at a site keeps track of local read-only transactions. Periodically, these modules communicate with each other to determine the versions of data that can be discarded. A simple propagation scheme can be used for this purpose [22], [13].

Another advantage of the modular version control scheme is that it can be easily integrated with databases that have both multiple versions as well as replication [14]. We are
currently developing a prototype database system that has both replication as well as multiple versions. The advantage of our approach is that the availability of read-only transactions is enhanced significantly [18].

VII. CONCLUSIONS

In this paper, we demonstrated that it is possible to decouple the version control from the concurrency control for multiversion databases. This modularization leads to an elegant and uniform interface between the components. The versatility of the interface is demonstrated by the ease and simplicity with which multiple conflict-based concurrency control protocols can be accommodated. An additional benefit of the decoupling is that read-only transactions undergo no concurrency control, and therefore, no overhead associated with related synchronization. Consequently, concurrency control processing of read-write transactions is completely independent of the read-only transaction processing. The decoupling and modularity of the mechanism also simplifies the extension of the version control mechanism to a distributed environment.

REFERENCES


Dr. Agrawal received the B.E. degree from Birla Institute of Technology and Science, Pilani, India in 1978 and the M.S. degree in computer science from the State University of New York at Stony Brook in 1984 and 1987, respectively. He is currently an Assistant Professor in the Department of Computer Science at the University of California at Santa Barbara. His research interest include design of algorithms for concurrent and fault-tolerant systems.

Dr. Agrawal is a member of the Association for Computer Machinery.

Soumitra Sengupta received the B.S. degree in electronics engineering from BITS, Pilani, India in 1980, and the M.S. and Ph.D. degrees in computer science from the State University of New York at Stony Brook in 1984 and 1987, respectively. He is currently an Associate Research Scientist in the Computer Science Department at Columbia University. His current research interests include design and development of large-scale software systems, databases, heterogeneous network and application management, and medical information systems.