Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control

Soumitra Sengupta[1]
Columbia University

Divyakant Agrawal[2]
UC, Santa Barbara

February 1989
Technical report No. CUCS-426-89

Abstract

In this paper we propose a version control mechanism 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. An interesting feature of our framework is that the execution of read-only transactions becomes completely independent of the underlying concurrency control implementation. Also, algorithms with the version control mechanism have several advantages over most other multiversion algorithms.

[1]Department of Computer Science, Columbia University, New York, NY 10027. The author was supported by the New York State under grant number NYSSTF CAT (88)-5.

[2]Department of Computer Science, University of California, Santa Barbara, CA 93106. The author was supported by the NSF under grant number CCR-8809387.

1 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 results 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 timestamp ordering [14] 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 then can 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 protocols, *viz.* two-phase locking, timestamp ordering, and optimistic concurrency control protocols [9, 14, 12]. One of the major advantages of the 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 [14]. 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 2, we briefly discuss other multiversion protocols and indicate their shortcomings. We present the formal model for correctness in multiversion databases in Section 3. In Section 4, we propose our version control mechanism and demonstrate how it can be integrated with the two-phase locking and timestamp ordering protocols. Correctness of these protocols is argued in Section 5. We discuss our results in Section 6, and concluding remarks appear in Section 7.

2 Multiversion Algorithms

Numerous concurrency control algorithms have been proposed for multiversion databases [14, 15, 4, 7, 8, 17]. 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 timestamp ordering was introduced by Reed [14]. The main advantage of this scheme is that read requests are never rejected. The algorithm can be viewed as an extension of the timestamp ordering protocol [4]. 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 operations 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 the database, distributed read-only transactions require two-phase commit protocol for their atomic commitment. Thus, execution of read-only transactions in this protocol has 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. 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 some additional information is associated with every read-only transaction. One is a *start timestamp* 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 a largest version of an object smaller than the start timestamp of the transaction, and ensuring that 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 the 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 [17]. We will discuss only one of them since others are primarily intended to integrate garbage-collection algorithms efficiently. The protocol which uses timestamps and initiation [17] 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 timestamps 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 inter-related 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 version control is that it allows simple and uniform integration with the standard concurrency control protocols such as timestamp 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, 16].

3 The Model

A *database* consists of a set of *objects*, and users interact with the database system by invoking *transaction* programs. A transaction T_i is an ordered pair $(\Sigma_i, <_i)$, where Σ_i is the set of operations in T_i, and $<_i$ indicates the execution

order of those operations. Read or Write operations executed by T_i on an object x are denoted by $r_i[x]$ and $w_i[x]$ respectively. We assume that there is at most one $r_i[x]$ and at most one $w_i[x]$ in Σ_i. Furthermore, if $r_i[x]$ and $w_i[x]$ are both in Σ_i then $r_i[x] <_i w_i[x]$. The execution of a transaction must appear atomic, *i.e.*, a transaction T_i terminates with either a *commit*, c_i, or an *abort*, a_i, operation. The commit of a transaction results in all its changes being applied to the database; the abort results in the changes being discarded.

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 Σ is the set of all operations executed by transactions in T, and $<_H$ indicates the execution order of those operations.

3.1 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. $\nexists w_k[x] \in H$ 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, and
3. they have the same *final writes* for each object x.

A *serial* history H_S is a total order such that 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 [13]. 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, and

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, 4].

3.2 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 $x_i, x_j, \ldots$, where the subscript is the (monotonically increasing) *version number* 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 destroyed.

A *multiversion* (MV) history H 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 $r_i[x]$ into $r_i[x_j]$, 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, are equivalent if they have the same operations. An MV history is *one-copy 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-copy serializable or not. To determine if an MV history is one-copy 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 $\ll_x$) on all transactions that write x, and

2. For each object x, if T_j *reads* x *from* T_i and if $T_i \ll_x T_k$, then $MVSG(H)$ has an edge from T_j to T_k (*i.e.*, $T_j \rightarrow T_k$); otherwise, if $T_k \ll_x T_i$, 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-copy serializable if $MVSG(H)$ is acyclic [5, 6].

4 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 *lock-point* [13]. A transaction number in timestamp ordering simply corresponds to the logical timestamp 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 two-phase locking and timestamp ordering protocols. The multiversion algorithm with version control and optimistic concurrency control appears in [1, 2] and, hence, is not presented in this paper.

4.1 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 actions.

2. Read-write Transactions: Transactions which update the state of the database, and therefore, execute at least one write action.

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 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 copy of an object exists in the database. The read operation is carried out by reading the most recent version of the object, and 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 *transaction number counter, tnc.*

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 choosing a value of *tnc* such that all transactions with transaction numbers smaller than the chosen value have completed. We can use this value to assign a number to the read-only transactions when they begin execution. This is called the start number of a transaction T, $sn(T)$. 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 above. This is precisely the source of complication in multiversion two-phase locking as proposed in [7] and [17]. A transaction T is assigned $tn(T)$ at the beginning in multiversion timestamp 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 *tnc* cannot be

used to assign start numbers to the read-only transactions. In our scheme we employ another monotonically increasing counter called *visible transaction number counter* (*vtnc*). Intuitively, the value of *vtnc* controls the visibility of the versions of data objects to the read-only transactions. Hence, *vtnc* serves the purpose of assigning start numbers to the read-only transactions. Unlike *tnc*, which is incremented when a transaction is assigned a transaction number, *vtnc* may be incremented 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, *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:

Transaction Ordering Property. The value of *tnc* at all times is the smallest number such that all transactions T, which either are active and unassigned or will arrive later, will have $tn(T) \geq tnc$.

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

Additionally, the values of the two counters must be such that $vtnc < tnc$ at all times.

The code related to the version control mechanism is illustrated in the module *VersionControl* in Figure 1. It has four entry procedures: $VCstart()$, $VCregister(T, status)$, $VCdiscard(T)$, and $VCcomplete(T)$. Also, this module maintains three data-structures related to version control: *tnc*, *vtnc*, and $VCQueue$. $VCQueue$ is an ordered list of all transactions that have been assigned 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.

```
MODULE VersionControl
PERSISTENT DATA tnc, vtnc : COUNT;
                VCQueue : QUEUE;

    PROCEDURE VCstart(): COUNT
        RETURN(vtnc);
    END VCstart;

    PROCEDURE VCregister(T : TransactionDesc; status : Status)
        tn(T) ← tnc++;
        Allocate entry E(T);
        E(T).type ← status;
        E(T).num ← tn(T);
        Insert(VCQueue, E(T));
    END VCregister;

    PROCEDURE VCdiscard(T : TransactionDesc);
        Discard(VCQueue, E(T));
    END VCdiscard;

    PROCEDURE VCcomplete(T : TransactionDesc);
        E(T).type ← "complete"
        WHILE (Head(VCQueue).type = "complete") DO
            vtnc ← Head(VCQueue).num;
            Delete(Head(VCQueue));
        END;
    END VCcomplete;
END.
```

Figure 1: The Version Control Module

The entry procedure *VCstart*() is invoked by a read-only transaction to obtain its start number. The entry procedure *VCregister*(T, "*active*") is invoked by a read-write transaction T at the time when its serialization order has been determined. Thus, in timetsamp ordering, this procedure is invoked right at the beginning, where as in two-phase locking this procedure is invoked when T reaches its lock point. The entry procedure *VCdiscard*(T) is invoked if T is aborted. On the other hand, if T commits, it invokes *VCcomplete*(T) after updating the database. Note that the procedure *VCcomplete*(T) enforces the transaction visibility property. That is, it sets the current value of *vtnc* to $tn(T)$ if all transactions with smaller transaction number have completed. Otherwise, the increment of *vtnc* is delayed.

4.2 Read-only Transactions

The execution of read-only transactions is shown in Figure 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 in the beginning. Afterwards, their existence remains transparent to both the concurrency and version control modules, and therefore, there is almost negligible overhead associated with read-only transactions in our scheme.

Action Invocation	Action Execution
begin(T)	$sn(T) \leftarrow VCstart()$
	$tn(T) \leftarrow sn(T)$ /* for proving correctness */
.	.
.	.
.	.
read(x)	return x_j with largest version $\leq sn(T)$
.	.
.	.
.	.
end(T)	ϕ

Figure 2: Execution of Local Read-only Transactions

A read-only transaction, T, begins its execution by obtaining its start num-

ber, $sn(T)$, from the version control module. Note, that $sn(T)$ for read-only transactions is also $tn(T)$. 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 $sn(T)$. Barring the unavailability of an appropriate version to read due to garbage-collection of old version, a read request of T is never rejected.

4.3 Version Control with Timestamp Ordering

Since the serial order of transactions in timestamp ordering is determined *a priori*, read-write transactions are assigned a transaction number before they begin execution. Figure 3 illustrates the execution of read-write transactions in a timestamp ordering protocol integrated with version control. The procedure *VCregister*(T,"*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 *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 non-serializable execution of read-only transactions. Note that in timestamp ordering protocol $sn(T)$ is the same as $tn(T)$.

Recall that in timestamp ordering protocols [14, 4], each version of an object x has a unique write timestamp, $w\text{-}ts(x)$, which records the transaction number of the transaction that created this version of x. The most recent version of x additionally has a read timestamp, $r\text{-}ts(x)$, which records the timestamp 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 timestamps of the most recent version of x. This conflict check is required in order to serialize transactions in their timestamp 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.

Action Invocation	Action Execution
$begin(T)$	$VCregister(T, \text{"}active\text{"})$
	$sn(T) \leftarrow tn(T)$
.	.
.	.
.	.
$read(x)$	$r\text{-}ts(x) \leftarrow MAX(r\text{-}ts(x), tn(T))$
	return x_j with largest version $\leq sn(T)$
	/* may be delayed due to the pending writes as per TO protocol*/
.	.
.	.
.	.
$write(y)$	IF $r\text{-}ts(x) > tn(T) \vee w\text{-}ts(x) > tn(T)$ THEN $abort(T); VCdiscard(T)$
	create x_j with version $tn(T)$
	$w\text{-}ts(x) \leftarrow MAX(w\text{-}ts(x), tn(T))$
.	.
.	.
.	.
$end(T)$	$commit(T)$
	perform database updates; clear pending read actions;
	$VCcomplete(T)$

Figure 3: Execution of Local Read-write Transactions in Timestamp Ordering

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 $w\text{-}ts(x)$ such that $sn(T) \geq w\text{-}ts(x)$. 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 $VCQueue$ 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 only for active and unaborted transactions.

4.4 Version Control with Two-phase Locking

In a two-phase locking protocol, the serial order of read-write transactions correspond 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 a transaction, T, signals a completion of its execution phase 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 Figure 4.

Action Invocation	Action Execution
$begin(T)$	$sn(T) \leftarrow \infty$ /* for uniformity */
.	.
.	.
.	.
$read(x)$	$r\text{-}lock(x)$
	/* may wait due to write locks as per 2PL protocol */
	return x_j with largest version $\leq sn(T)$
.	.
.	.
.	.
$write(y)$	$w\text{-}lock(y)$
	/* may wait due to other locks as per 2PL protocol */
	create y_j with version ϕ
.	.
.	.
.	.
$end(T)$	$VCregister(T,$"$active$"$)$
	$commit(T)$
	perform database updates with version number $tn(T)$
	clear locks
	$VCcomplete(T)$

Figure 4: Execution of Local Read-write Transactions in Two-phase Locking

A read-write transaction, T, in two-phase locking scheme always reads the latest version of objects. Hence, for the purpose 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 non-committed version of y after it acquires a write lock on y. The difference from the timestamp ordering protocol, however, is that T does not have an assigned $tn(T)$ as yet. But this is not a problem because this protocol will not allow any other transaction to read 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 [17]), 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.

5 Correctness

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

5.1 Proof of Version Control with Timestamp Ordering

The following lemmas state certain properties of this protocol. We will use these properties to prove that the protocol is one-copy 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 timestamp ordering protocol, several read-only transactions in this scheme may be assigned the same transaction number. However, this property 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. □

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 action $r_k[x_j]$ 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 timestamp concurrency control protocol (through the version control action $VCRegister$) assigns $tn(T_k)$ before T_k executes any of its action. 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 action. 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 final 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 $r_k[x_j]$ and $w_i[x_i] \in H$, $i \neq j$, one of the following conditions must hold:

(i) $tn(T_i) < tn(T_j)$, or

(ii) $tn(T_k) < tn(T_i)$, or

(iii) $i = k$ 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 I. T_k is a read-write transaction. Let us assume $i \neq k$. From Lemma 2, $tn(T_j) < tn(T_k)$. Furthermore, the timestamp concurrency control protocol would reject an operation $w_i[x_i]$ if $tn(T_j) < tn(T_i) < tn(T_k)$. Hence, 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 3).
Case II. T_k is a read-only transaction. From Lemma 2, $tn(T_j) \leq tn(T_k)$. Since T_k 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 $tn(T_j) < tn(T_i) \leq tn(T_k)$. Therefore, there are only two possibilities: either $tn(T_i) < tn(T_j)$, 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-copy serializability. The theorem is an extension of the theorem for the multiversion timestamp ordering [6].

Theorem 1 Version control with timestamp ordering guarantees serializable execution of transactions.

Proof. We define the version order as the transaction number of the creators of version, *i.e.*, $x_i \ll_x x_j$ if and and only if $tn(T_i) < tn(T_j)$.

Let H be a history produced by the version control with timestamp ordering protocol. We will prove that $MVSG(H)$ is acyclic by showing that for each edge $T_i \rightarrow T_j$ in $MVSG(H)$, $tn(T_i) < tn(T_j)$.

Both T_i and T_j are read-write transactions since $MVSG(H)$ has only those transactions that interact with the concurrency control component. Recall that $MVSG(H)$ includes edges in $SG(H)$ and additional version order edges. Consider the edges in $SG(H)$. Each edge $T_i \rightarrow T_j$ 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, $tn(T_i) < tn(T_j)$. Hence, if there is an edge $T_i \rightarrow T_j$ in $SG(H)$, $tn(T_i) < tn(T_j)$.

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

1. $x_i \ll_x x_j$, which implies $T_i \rightarrow T_j$ is in $MVSG(H)$, and
2. $x_j \ll_x x_i$, which implies $T_k \rightarrow T_i$ is in $MVSG(H)$.

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

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

5.2 Proof of Version Control with Two-phase Locking

The proof of this algorithm is similar to the previous one, and we omit the proof for lack of space. A complete proof appears in [3].

6 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 timestamp 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 [17].

In order to achieve the advantages mentioned above, 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 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 unacceptable, 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 *concurrency* at the expense of *currency* [10]. Such transactions can be dealt with by executing them as pseudo read-write transactions.

A complete description of a version control mechanism for a distributed database appears in [3]. Our formulation of distributed version control is not very complex, and it guarantees global serializability of read-only transactions. Since each database site in a distributed environment maintains its own counters (*tnc* and *vtnc*) and its own queue (*VCQueue*), care must be taken to ensure correctness. However, once we ensure that there is only one start num-

ber associated with a read-only transaction and only one transaction number for every read-write transaction, the extension of centralized version control to a distributed one is quite straightforward.

No description of a multiversion algorithm is complete unless some attention is given to the process of garbage collection of old and unnecessary version of data. In our scheme, a garbage collection algorithm, which keeps the information about read-only transactions, can be easily integrated. 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 *vtnc*. 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.

7 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 controls can be accommodated. An additional benefit of the decoupling is that read-only transactions undergo no concurrency control, and therefore, have no overhead associated with related synchronization. Consequently, concurrency control processing of read-write transactions is completely independent of the read-only transaction processing.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed multiversion optimistic concurrency control for relational databases. In *The 31st IEEE Computer Society International Conference*, March 1986.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Journal of Distributed Computing, Springer-Verlag*, 2(1):45–59, January 1987.

[3] D. Agrawal and S. Sengupta. Distributed version control. Technical report, Department of Computer Science, University of California at Santa Barbara & Department of Computer Science, Columbia University, 1989. In preparation.

[4] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency control: Theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

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

[7] A. Chan, S. Fox, W. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the ACM-SIGMOD*, pages 184–191, July 1982.

[8] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, February 1985.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[10] H. Garcia-Molina and G. Weiderhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[11] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[12] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[13] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[14] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1978.

[15] R. E. Stearns and D. J. Rosenkrantz. Distributed database concurrency control using before-values. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 74–83, June 1981.

[16] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transaction on Database Systems*, 4(2):180–209, June 1979.

[17] W. E. Weihl. Distributed version management of read-only actions. *IEEE Transactions on Software Engineering*, 13(2):55–64, January 1987.