Using Multiversion Data for Non-interfering Execution of Write-only Transactions *

D. Agrawal V. Krishnaswamy
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

A modular version control mechanism is developed for executing write-only transactions with minimal interference to read-write transactions using multiversion data. The execution of write-only transactions is completely independent of the underlying concurrency control protocol. The version control mechanism provides the versatility of using any conflict-based concurrency control protocol for read-write transaction synchronization. An integrated version control mechanism is presented in which both read-only and write-only transactions are handled symmetrically, and are independent of the concurrency control mechanism. In addition, there is negligible version control related overhead for executing read-only and write-only transactions. Our approach of non-interfering execution of write-only transactions is particularly useful in database systems consisting of abstract data objects where blind-write operations are dominant.

1 Introduction

Multiple versions of data are used in database systems to increase concurrency and to support recovery from transaction and system failures. The higher concurrency results since out-of-order read requests can be processed by reading appropriate, older versions of data. Recovery from transaction and system failures is considerably simplified in multiversion databases since before-images of write operations need not be maintained explicitly. When failures occur, versions created by failed transactions can be simply discarded. In addition, the use of multiversion data in the form of operation histories has been advocated by several researchers to implement abstract data types efficiently [9, 14].

In most protocols for multiversion data, the main emphasis is always directed to the execution of *read-only* transactions. Read-only transactions in most multiversion schemes are executed with negligible concurrency control overhead [12, 6, 13, 4]. Most schemes partition the transactions into two classes: read-only transactions and read-write transactions. As a result of this classification, the execution of read-only transactions can be made independent of the underlying concurrency control protocol [4]. This is highly desirable since read-only transactions can be executed without incurring any synchronization overhead and furthermore with no interference to read-write transactions.

In the case of multiversion databases that use operation histories as a means of object representation in the database, another class of transactions becomes prominent in the system. These transactions are often referred to as *blind-write* transactions [8]. Such transactions consist of operations that do not observe the states of the objects; instead they "blindly" modify the object states. For example, a deposit transaction on an account object does not observe the state of the account object history. It only needs to append the deposit operation with a specified input parameter to the history. Similarly, in an airlines reservation system, which stores operation histories, reservation and cancellation transactions can be executed as blind decrement and increment transactions, respectively. Thus, in such systems it seems natural that transactions should be classified into three categories: read-only, read-write, and write-only[1] transactions. After classifying the transactions into these categories, the question arises if multiversion

*This research is supported by the NSF under grant number IRI-9004998, by the University of California and Xerox Corporation under grant number MICRO 88-204, and by the University of California and Rockwell International under grant number MICRO 90-005.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0098...$1.50

[1] We are using the term *write-only* instead of *blind-write* for the sake of uniformity.

data can also be used to eliminate or minimize the interference between read-write and write-only transactions. None of the protocols for multiversion databases address this problem. In many database applications related to spatial data, information collected by outside sources is often appended to the database. Such addition of information can be performed asynchronously by write-only transactions, eliminating the interference with other transactions.

A separate treatment of write-only transactions, however, has been considered in the protocols for replicated databases [8]. This can be accomplished in replicated databases since multiple versions of data objects temporarily exist due to the different values of the replicas. Our earlier work on modularization of the version control component in multiversion databases was motivated by the modularity of replica control and concurrency control protocols in replicated databases. In this paper, we are once again motivated by replicated data management protocols that permit non-interfering execution of write-only transactions.

In this paper, we present a version control mechanism that minimizes the interference between read-write and write-only transactions. Also, the execution of write-only transactions becomes completely independent of the underlying concurrency control protocol. By combining the new mechanism with our modular version control mechanism we develop a protocol for multiversion databases that has the following properties. The treatment of read-only and write-only transactions is symmetric and is independent of the underlying concurrency control mechanism. Also, both read-only and write-only transactions cause minimal interference to the read-write transactions in the system. In addition, there is negligible version control related overhead with the read-only and write-only transactions. The version control mechanism provides the versatility of using any conflict-based concurrency control protocol for read-write transaction synchronization. The main contribution of this paper is that it fills the void that is left by the existing algorithms for multiversion data.

The rest of the paper is organized as follows. In Section 2, we present the formal model for correctness in multiversion databases. In Section 3, we present the version control mechanism, and show how write-only transactions are executed under this mechanism. In this section, we also demonstrate how the version control mechanism can be integrated with the two-phase locking protocol. Correctness of the version control scheme with two phase locking is argued in Section 4. Next, in Section 5, we present an integrated version control mechanism in which read-only and write-only transactions are handled symmetrically. Concluding remarks appear in Section 6.

2 The Database Model

A *database* is a collection of *objects*. Users interact with the database by invoking *transactions*. A transaction is a totally ordered sequence of read and write operations that are executed atomically on the objects. A read (write) operation executed by a transaction T_i on object x is denoted as $r_i[x]$ ($w_i[x]$). The commit of a transaction results in all its changes being applied to the database, while the abort results in the changes being discarded. Finally, a transaction is assumed to be *correct*, i.e., it maps the database from one *consistent* state to another consistent state. We assume that if T_i both reads and writes object x, then $r_i[x] < w_i[x]$. Consider a set of transactions $\mathsf{T} = \{T_1, T_2, \ldots, T_n\}$. The execution of the transactions in T is modeled by a structure called a *history*. Formally, a history H over T is an ordered pair $(\Sigma, <_H)$ where Σ is the set of all operations executed by transactions in T, and $<_H$ reflects the order in which the operations were executed. A *serial* history is a totally ordered history such that for every pair of transactions T_i and T_j, either all operations executed by T_i precede all operations executed by T_j or vice versa. Two operations *conflict* if they both operate on the same object, and one of them is a write. A history H is *conflict serializable* [11] if there exists some serial history H_s over the same set of transactions, such that if op_1 and op_2 conflict and $op_1 <_H op_2$ then $op_1 <_{H_s} op_2$. 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 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 [7, 11]. A *concurrency control protocol* is one that ensures that all transaction executions are serializable.

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 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 represents a 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. 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.

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 versions of x, and
2. For each object x, if T_j *reads* x *from* T_i and if $x_i \ll_x x_k$, then $MVSG(H)$ has the edge $T_j \rightarrow T_k$; otherwise, if $x_k \ll_x x_i$, then $MVSG(H)$ has the edge $T_k \rightarrow T_i$.

The additional edges are called *version order edges.* An MV history H is one-copy serializable if $MVSG(H)$ is acyclic [5].

3 The Version Control Mechanism for Write-only Transactions

In this section, we first give an overview of the version control scheme for implementing non-interfering execution of write-only transactions. Next, we present the version control module that defines the interface for read-write and write-only transactions. We then describe the execution of write-only transactions, which is independent of the underlying concurrency control mechanism. Finally, we present a multiversion algorithm in which version control is integrated with two phase locking [7] for executing read-write transactions.

3.1 Overview

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 serialization order. Furthermore, a read-write transaction T is assigned a transaction number $tn(T)$ which is unique and corresponds to its position in the serialization order. That is, if T_1 precedes T_2 in the serialization 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 modified 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* [11]. A transaction number in timestamp ordering simply corresponds to the logical timestamp of the transaction.

We also assume that all transactions in the system can be classified *a priori* into two categories:

1. Write-only Transactions: Transactions which do not observe the state of the database, and therefore, do not execute any read actions.
2. Read-write Transactions: Transactions which observe, and (possibly) update the state of the database, and therefore, execute at least one read 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 an 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 versions of the objects written by a read-write transaction T is its transaction number $tn(T)$. In order to assign these numbers to transactions the version control mechanism maintains a monotonically increasing counter called *transaction number counter, tnc.*

The execution of write-only transactions can be implemented by drawing the analogy from the treatment of read-only transactions in multiversion algorithms. Recall that most multiversion algorithms [6, 13, 4] eliminate the interference of read-only transactions with other transactions by serializing the read-only transactions in the *past.* That is, read-only transactions are given a snapshot of the database that existed in the recent past. This was accomplished in the modular version control mechanism [4] by introducing a counter *vtnc*, which trails *tnc*. The value of *vtnc* is such that all transactions with smaller transaction numbers have completed. Read-only transactions are serialized with respect to the value of this counter.

By symmetry, the interference of write-only transactions with other transactions can be eliminated by serializing the write-only transactions in the *future* with respect to active read-write transactions. A trivial solution would be to assign all write-only transactions a transaction number of "infinity" and not allow other transactions to observe object states corresponding to the write-only transactions. Since no transaction observes the effect of write-only transactions, the read-write transactions need not be synchronized with respect to the write-only transactions. The solution, however, is analogous to allowing read-only transactions to only see the initial state of the database.

A more appropriate approach to implement non-interfering execution of a write-only transaction would be to serialize the write-only transaction after all currently active read-write transactions. That is, a write-only transaction must be committed with a transaction number that is larger than the transaction numbers of all active read-

write transactions. However, in many concurrency control schemes, e.g., two phase locking, active transactions may execute without being assigned a transaction number. Thus it becomes difficult to assign transaction numbers to write-only transactions if there exist active read-write transactions which have not yet been assigned a transaction number. We therefore introduce another transaction counter in addition to *tnc* and call it *future transaction number counter, ftnc*, since this counter is used to serialize write-only transactions in the immediate future.

A data structure, *VCQueue*, is maintained by the version control mechanism to capture the history of the execution of transactions in the system. *VCQueue* is used to make the versions created by transactions visible in the order of their serialization. Figure 1 illustrates a snapshot of *VCQueue*. The counter *tnc* has a value equal to the transaction number to be assigned to the first transaction in the first active set in *VCQueue*. In the example shown in Figure 1, the first active set in *VCQueue* is the set A. The counter *ftnc* has a value larger than the transaction numbers of all active and completed transactions (after the current set F in Figure 1). When a read-write transaction arrives, it is added to the current set of active read-write transactions in *VCQueue* (set F in Figure 1) and *ftnc* is incremented by one. When an active read-write transaction's position in the serialization order is determined (by the concurrency control protocol used), a transaction number is assigned to it (either from *tnc* or from the transaction number of a transaction immediately preceding it in *VCQueue*). It is then removed from its set entry and inserted as a transaction entry (an entry consisting of this transaction only) in *VCQueue*. For example, in Figure 1, C could be one such transaction entry, which belonged to set D before its position in the serialization order is known. The transaction numbers of transactions are in correspondence with their ordering within *VCQueue*. When a write-only transaction arrives it is added to *VCQueue* after the current set of active read-write transactions, i.e., after set F in Figure 1 and *ftnc* is incremented by one. A *transaction entry* with *status* "complete" is removed from *VCQueue* when all *entries* preceding it are "complete" and have been removed. If a read-write transaction gets aborted, then its corresponding entry is discarded from *VCQueue*. Write-only transactions are never aborted by our protocol.

In the following section, we describe the relationship of the two counters *tnc* and *ftnc*, and show how these counters can be used to execute write-only transactions asynchronously, and then we present the version control module.

3.2 Version Control

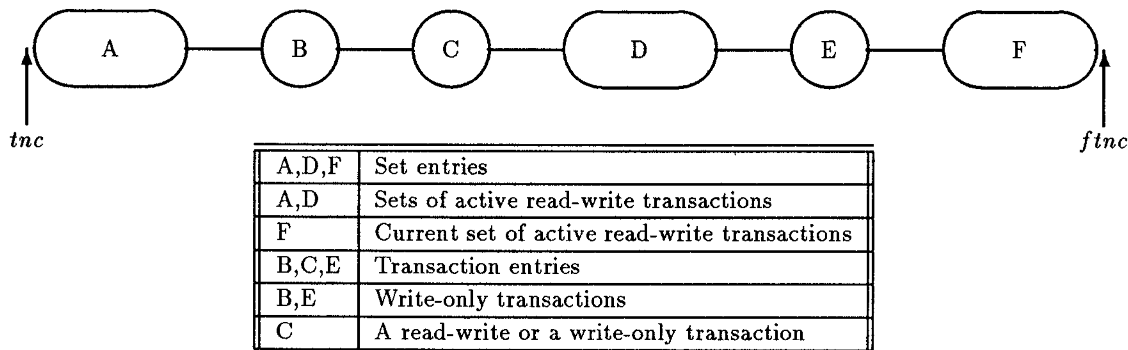
As in the case of read-write transactions, the version number of the versions of the objects written by a write-only transaction W is its transaction number $tn(W)$. To assign transaction numbers to write-only transactions, the version control mechanism maintains a monotonically increasing counter, *ftnc*. If we make the versions of data objects written by a write-only transaction not to be visible to all active transactions that started before it, and visible to all future transactions that start after it, then we can easily serialize the write-only transactions in the system.

This is accomplished by first assigning a start number, $sn(T)$ for each read-write transaction T to be infinity. If a write-only transaction W is installed, then $sn(T)$ for all active transactions T in the current set, that started before W and after the previous write-only transaction (if any), are reset to $tn(W) - 1$. Any read operation of T is carried out by reading the version with the largest version number less than or equal to $sn(T)$. It can be informally argued that W succeeds all active read-write transactions that started before it, and precedes any future read-write transactions that start after it.

The two counters *tnc* and *ftnc* are maintained as follows: a read-write transaction is assigned $tn(T)$ at the earliest point in time, when its position in the serialization order can be determined. This point depends on the concurrency control protocol that is used to synchronize the read-write transactions. For example, in the version control scheme with two phase locking $tn(T)$ is assigned when T reaches its lock point, while in the version control scheme with time stamp ordering, it is done when the transaction begins its execution. A write-only transaction W is assigned its transaction number, when its writes are committed, by using *ftnc*. Since we want all active read-write transactions that started before W to be serialized before W, *ftnc* is used to assign the transaction number for W. The counter *ftnc* is then incremented by one so that it is larger than all active and completed transactions, including W. A read-write transaction T is assigned a transaction number that lies between the transaction numbers of some consecutive pair of write-only transactions, between whose installations T started. Thus any read-write transaction T serializing between two consecutive write-only transactions, W and W', obtains a transaction number $tn(T)$, such that $tn(W) < tn(T) < tn(W')$. Either *tnc* or the transaction number of the transaction in the immediately preceding *entry* in *VCQueue* is used for this assignment. The counter *tnc* is then incremented so that it corresponds to the first active set of transactions in *VCQueue*. Thus, 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 largest 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 *ftnc* at all times is such that all active and completed transactions T, have transaction numbers $tn(T) < ftnc$.



A,D,F	Set entries
A,D	Sets of active read-write transactions
F	Current set of active read-write transactions
B,C,E	Transaction entries
B,E	Write-only transactions
C	A read-write or a write-only transaction

Figure 1: A Snapshot of *VCQueue*

Additionally, the values of the two counters are such that $tnc \leq ftnc$ at all times.

The interface to the version control component is illustrated in the module *VersionControl* in Figure 3. It has five entry procedures: $VCbegin(T)$, $VCregister(T)$, $VCinstall(W)$, $VCdiscard(T)$, and $VCcomplete(T)$. Also, this module maintains three data structures related to version control: *tnc*, *ftnc*, and *VCQueue*. Since the procedures access and update these shared structures, they must be executed in a critical section. *VCQueue* is an ordered list of type *entry*. An *entry* is one of two types, as described in Figure 2. The first type of *entry* is a *transaction entry*, where the *element* field contains the transaction descriptor, the *type* field is "transaction", and the *num* field contains the transaction number. The *status* field of this *entry* is "complete" or "active" depending on whether the transaction has invoked the procedure *VCcomplete*. The second type of *entry* is a *set entry*, where the *element* field contains a set of transaction descriptors, the *type* field is "set", the *num* field contains the transaction number of the youngest write-only transaction preceding it, and the *status* field is "active".

There are six operations defined on *VCQueue*. The *Create* operation creates an empty *VCQueue*. The *Insert* operation inserts a *transaction entry* in *VCQueue* according to its transaction number while maintaining the order of *entries* in *VCQueue*. The *Discard* operation deletes a specific *entry*. The *Head* operation returns the first *entry* of *VCQueue*. The *Delete* operation removes as well as returns the first *entry* of *VCQueue*, and the *Append* operation adds an *entry* to the tail of *VCQueue*.

The *CurrentSet* points to an *entry* that consists of the set of all active read-write transactions that have started after the youngest write-only transaction. Initially, *CurrentSet* points to a *set entry* whose *element* field is ϕ, *type* field is "set", *num* field is 0, and *status* field is "active". Read-write transactions are added to the *set entry* that *CurrentSet* points to in *VCQueue*, at the start of their execution. When the position of a read-write transaction in the serialization order is determined, it is removed from the *set entry* that it belongs to, and is inserted into *VCQueue* as a *transaction entry*.

In a serial order, a write-only transaction succeeds all active read-write transactions that have started before it. Hence it is inserted in *VCQueue* as a *transaction entry* after the *set entry* pointed to by *CurrentSet*. A *transaction entry* with *status* "complete" is removed from *VCQueue* when all *entries* preceding it are "complete" and have been removed. If a read-write transaction gets aborted, then it is removed from *VCQueue*. Write-only transactions are never aborted by our protocol.

The entry procedure $VCbegin(T)$ is invoked by a read-write transaction T to obtain its start number and to be added to the *entry* pointed to by *CurrentSet*; $tn(T)$ at this time is assigned a value that is a lower bound for the final value of $tn(T)$. The need for this assignment will become clear later. $VCregister(T)$ is invoked by a read-write transaction T at the time when its position in the serialization order is determined, to obtain its transaction number $tn(T)$. Thus, in the version control scheme with two phase locking this procedure is invoked when T reaches its lock point, and in the version control scheme with timestamp ordering, when T begins its execution. 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. $VCinstall(W)$ is invoked when a write-only transaction W completes its execution. Note that the procedures $VCinstall(W)$ and $VCbegin(T)$ enforce the transaction visibility property. That is, they set the current value of *ftnc*. $VCinstall(W)$ also sets $tn(W)$ and resets the start numbers for all active read-write transactions in the current set, that started before

```
TYPE
    entry : RECORD
        num : INTEGER;
        type : {"set", "transaction" };
        status : {"active", "complete" };
        element : union
            A : transactiondesc;
            B : set of transactiondesc;
        end;
    end;
```

Figure 2: Description of the Type *entry*

```
MODULE VersionControl
PERSISTENT DATA
    tnc : COUNT default 1; ftnc : COUNT default 1;
    VCQueue : QUEUE of entry;
    CurrentSet : POINTER TO entry default NIL;

    PROCEDURE VCbegin(T : transactiondesc);
        ftnc++; sn(T) ← ∞;
        CurrentSet↑.element ← CurrentSet↑.element ∪ {T};
        tn(T) ← CurrentSet↑.num; E(T) ← CurrentSet;
    END VCbegin;

    PROCEDURE VCregister(T : transactiondesc);
        IF ∃E(T') immediately preceding E(T) THEN
            tn(T) ← tn(T') + 1;
        ELSE
            tn(T) ← tnc++;
        END; (* IF *)
        E(T) ↑ .element ← E(T) ↑ .element \ {T}
        IF E(T) ≠ CurrentSet ∧ E(T) ↑ .element = φ THEN
            Discard(VCQueue, E(T));
        END; (* IF *)
        Allocate entry E(T); E(T) ↑ .type ←"transaction"; E(T) ↑ .element ← T;
        E(T) ↑ .status ← "active"; E(T) ↑ .num ← tn(T); Insert(VCQueue, E(T));
        Increment tnc until it encounters a set entry or the end of VCQueue;
    END VCregister;

    PROCEDURE VCdiscard(T : transactiondesc);
        IF E(T) ↑ .type ="transaction" THEN
            Discard(VCQueue, E(T));
        ELSE
            E(T) ↑ .element ← E(T) ↑ .element \ {T}
            IF E(T) ≠ CurrentSet ∧ E(T) ↑ .element = φ THEN
                Discard(VCQueue, E(T));
                Increment tnc until it encounters a set entry or the end of VCQueue;
            END; (* IF *)
        END; (* IF *)
    END VCdiscard;

    PROCEDURE VCcomplete(T : transactiondesc);
        E(T) ↑ .status ←"complete";
        WHILE (Head(VCQueue) ↑ .status ="complete") DO
            Delete(VCQueue);
        END; (* WHILE *)
    END VCcomplete;
```

Figure 3: The Version Control Module for Write-only Transactions (contd. on the next page)

```
MODULE VersionControl (contd.)
    PROCEDURE VCinstall(W : transactiondesc);
        tn(W) ← ftnc + +;
        Allocate entry E(W); E(W) ↑ .type ←"transaction"; E(W) ↑ .element ← W;
        E(W) ↑ .status ←"complete"; E(W) ↑ .num ← tn(W); Append(VCQueue, E(W));
        IF CurrentSet ↑ .element = φ THEN
            Discard(VCQueue, CurrentSet);
        ELSE
            ∀T' ∈ CurrentSet ↑ .element, sn(T') ← tn(W) − 1;
        END; (* IF *)
        Allocate entry CurrentSet; CurrentSet ↑ .type ← "set"; CurrentSet ↑ .element ← φ;
        CurrentSet↑.status ←"active"; CurrentSet↑.num ← tn(W);
        Append(VCQueue, CurrentSet);
        Increment tnc until it encounters a set entry or the end of VCQueue;
        Install the writes of W with version number tn(W);
    END VCinstall;
BEGIN
    Create(VCQueue);
    Allocate entry CurrentSet; CurrentSet↑.type ← "set"; CurrentSet↑.element ← φ;
    CurrentSet↑.status ←"active"; CurrentSet↑.num ← 0;
    Append(VCQueue, CurrentSet);
END VersionControl.
```

Figure 3: The Version Control Module for Write-only Transactions (contd. from the previous page)

W and after the previous write-only transaction (if any), to $tn(W) - 1$. The counter tnc is incremented in the procedures $VCregister(T)$, $VCdiscard(T)$, and $VCinstall(W)$ to maintain the transaction ordering property.

3.3 Write-only Transactions

The execution of write-only transactions is shown in Figure 4. The left column shows the action of a write-only transaction W, and the right column illustrates the resulting execution of the same action. The execution of write-only transactions in our scheme is 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 when $end(W)$ is invoked. Therefore, there is almost negligible version control related overhead associated with write-only transactions.

Each procedure within the version control module has to be executed atomically, since the procedures access/update shared data structures. However, in order to allow for concurrent execution of different procedures of the version control module, the counter $ftnc$ is incremented by a predetermined value δ. The value δ is determined based on the ratio of the number of read-write transactions to the number of write-only transactions in the system, and on the arrival rate of read-write transactions. In this case, a write-only transaction can concurrently execute with read-write transactions.

3.4 Version Control with Two-phase Locking

In a two phase locking protocol, the serialization 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 position in the serialization order is uncertain. Therefore, in this version control scheme with two phase locking, a read-write transaction 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 $end(T)$.

A read-write transaction, T, in this scheme always reads the latest version of objects. Hence $sn(T)$ is initially 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 $x \leq sn(T)$ in the database. A write request from T for y results in obtaining a write lock on y. If the lock is not available, T is delayed. Otherwise, T creates a version of y, which gets a version number equal to $tn(T)$, when T commits.

Introduction of write-only transactions gives rise to the following anomaly. If W is a write-only transaction, and T_i and T_j are read-write transactions such that T_i starts before and T_j starts after W is installed, then irrespective of their lock-points, T_i should be serialized before T_j. Therefore the start numbers of read-write transactions should have a value which guarantees that they observe the ap-

Action Invocation	Action Execution
$begin(W)$	ϕ
...	...
$write(x)$	create a version of x;
...	...
$end(W)$	$VCinstall(W)$;

Figure 4: Execution of Write-only Transactions

Action Invocation	Action Execution
$begin(T)$	$VCbegin(T)$;
...	...
$read(x)$	$r\text{-}lock(x)$; /* may wait due to write locks */ $r_ts(x) = max(r_ts(x), tn(T))$; return x_j with largest version $\leq sn(T)$;
...	...
$write(y)$	$w\text{-}lock(y)$; /* may wait due to other locks */ if $r_ts(y) > sn(T)$ then $abort(T)$; $VCdiscard(T)$; else create a version of y; end;
...	...
$end(T)$	$VCregister(T)$; perform database updates with version number $tn(T)$; clear locks; $commit(T)$; $VCcomplete(T)$;

Figure 5: Execution of Read-write Transactions in Version Control with Two-phase Locking

propriate versions corresponding to their position in the serialization order. Also, future writes which need to be serialized earlier should be aborted if they invalidate any reads. We achieve this in the version control module by means of the following steps:

1. T_i is prevented from reading x_j written by T_j by setting $sn(T_i) < tn(T_j)$ when W is installed. Since the version number of x_j is greater than $sn(T_i)$, T_i cannot read x_j.

2. T_i is prevented from writing an object x after T_j has read x. This is accomplished by maintaining a read timestamp on x, $r_ts(x)$. Transaction T_j on reading x, updates $r_ts(x)$ to be greater than or equal to $tn(W')$, where W' is the youngest write-only transaction that precedes the *entry* containing T_j. The future write of T_i on x, on its arrival, finds that $r_ts(x) \geq tn(W') > sn(T_i)$, and hence T_i is aborted.

The version control scheme with two phase locking delineates write-only and read-write transactions completely. Since a write-only transaction execution is independent of the concurrency control protocol, it is unaffected by concurrent read-write transactions. 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-points. Since such transactions cannot have any pending lock requests, they cannot be involved in any deadlock cycle. The execution of a read-write transaction T in a two phase locking protocol integrated with version control is illustrated in Figure 5. The left column shows the action of a read-write transaction, and the right column illustrates the resulting execution of the same action.

4 Correctness

In this section we demonstrate that the version control scheme with two phase locking developed in the previous section guarantees serializability of all transactions.

The following lemmas state certain properties of this protocol for handling write-only transactions. We will use these properties to prove that the protocol is one-copy serializable. In what follows if the type of the *entry* is not specified, then the *entry* can be either a *transaction entry* or a *set entry*. The first lemma shows that transactions in this protocol are assigned unique transaction numbers. Note that the lemma holds for both read-write and write-only transactions.

Lemma 1 For each transaction T there is a unique transaction number $tn(T)$.
Proof. Appears in [3]. □

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

Lemma 2 For every $r_k[x_j] \in H$, $w_j[x_j] <_H r_k[x_j]$ and $tn(T_j) < tn(T_k)$.
Proof. Appears in [3]. □

The following lemma states that, when a transaction T reads an object x, it reads a version of x with the largest version number less than or equal to $sn(T)$. In addition, if another transaction later attempts to write x with a version number smaller than the read timestamp on x, $r_ts(x)$, then 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 $r_k[x_j] <_H w_i[x_i]$.

Proof. Appears in [3]. □

By using the above lemmas as formal specifications of the protocol, the following theorem demonstrates that the protocol guarantees one-copy serializability.

Theorem 1 The version control scheme with two phase locking guarantees serializable execution of transactions.

Proof. We define the version order $\ll_x$ for an object x as the total order on transaction numbers of the transactions creating versions of x, 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 scheme with two phase locking protocol. We will prove that $MVSG(H)$ is acyclic by showing that for each edge $T_i \rightarrow T_j$ in $MVSG(H)$, $tn(T_i) < tn(T_j)$.

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, $tn(T_i) < tn(T_j)$. 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 $w_i[x_i] \in H$ 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 on the transaction numbers of transactions involved in that cycle. Hence $MVSG(H)$ is acyclic. Thus by application of the serializability theorem for multiversion data [5], every history H produced by the version control scheme with two phase locking is one-copy serializable. □

5 Multiversion Databases: Read-only, Read-write, and Write-only Transactions

In this section, we show how our version control mechanism can be altered to implement non-interfering execution of read-only transactions in addition to write-only transactions. In this integrated mechanism, the execution of both read-only and write-only transactions is independent of the underlying concurrency control mechanism.

We assume that all transactions in the system can be classified *a priori* into three categories:

1. Read-only Transactions: Transactions which do not update the state of the database, and therefore, do not execute any write actions.
2. Write-only Transactions: Transactions which do not observe the state of the database, and therefore, do not execute any read actions.
3. Read-write Transactions: Transactions which observe and update the state of the database, and therefore, execute at least one read action and one write action.

Read-write and write-only transactions interact with the integrated version control mechanism as before. We now explain the interaction of read-only transactions with the integrated version control mechanism, and then show how read-only transactions are executed.

Read-only transactions can be serialized in the system if the versions of data objects that are visible to a read-only transaction R are such that no versions with version numbers less than $tn(R)$ are created by any active or future transactions. This is accomplished by assigning a start number $sn(R)$ to a read-only transaction R, such that all transactions with transaction numbers less than or equal to $sn(R)$ have completed. Any read operation of R is carried out by reading the version with the largest version number less than or equal to $sn(R)$. It can be informally argued that R is serialized according to $sn(R)$, i.e., it precedes all active and future read-write and write-only transactions, and all transactions that precede R are completed.

In order to assign start numbers to read-only transactions, 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. The value of *vtnc* may be incremented only when a transaction T completes. It will be left unchanged, if, at the time T completes, there is another transaction T' that is still active with $tn(T') < tn(T)$. This situation may arise 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 to read-only transactions according to the serialization order of transactions in the system. Hence, we can state the following property for *vtnc*:

> 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.

The values of *tnc*, *vtnc*, and *ftnc* are such that $vtnc < tnc \leq ftnc$ at all times. The detailed description of the integrated version control module that supports all three classes of transactions is presented in [3].

6 Discussion

Read-only transactions and write-only transactions in a multiversion algorithm with the version control mechanism do not interact with the concurrency control component and, therefore, the multiversion algorithm with version control does not have any synchronization overhead for read-only or write-only transactions. The version control interface also provides the versatility of using any conflict-based concurrency control protocol for read-write transaction synchronization. For example, the version control mechanism can also be easily integrated with timestamp ordering [12], optimistic concurrency control [10, 1] or with other two phase locking protocols [2]. One drawback is that write-only transactions may cause aborts of read-write transactions in the version control scheme with two phase locking. However, in the version control scheme with timestamp ordering, such a phenomenon does not occur.

An interesting contribution of this paper is that of using multiversion data to minimize interference between write-only transactions and read-write transactions. This feature has not been addressed in any existing protocols for multiversion databases. Also, read-only and write-only transactions are treated symmetrically, i.e., read-only transactions are serialized in the past and write-only transactions are serialized in the future using $vtnc$ and $ftnc$ respectively. The values of $vtnc$ and $ftnc$ are symmetrical in that for all active transactions T, $vtnc < tn(T)$, and $ftnc > tn(T)$. Our approach of non-interfering execution of write-only transactions is particularly useful in databases with abstract data objects. In such environments, the class of write-only transactions becomes dominant due to the existence of several blind-write operations, viz., deposit operation on a bank account object, enqueue operation on a queue object, increment and decrement operations on a counter object, and so on. Thus, if transactions involve only blind-write operations, they can be executed efficiently by using the mechanism described in this paper, eliminating the need for concurrency control for such transactions.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Optimistic Concurrency Control with reduced Rollback. *Distributed Computing, Springer-Verlag*, 2(1):45–59, January 1987.

[2] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990.

[3] D. Agrawal and V. Krishnaswamy. Using Multiversion Data for Non-interfering Execution of Write-only Transactions. Technical report, Department of Computer Science, University of California at Santa Barbara, CA 93106, 1991. TRCS 91-3.

[4] D. Agrawal and S. Sengupta. Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 408–417, May 1989.

[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] 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 International Conference on Management of Data*, pages 184–191, July 1982.

[7] 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.

[8] A. Heddaya. *Managing Event-based Replication for Abstract Data Types in Distributed Systems.* PhD thesis, Department of Computer Science, Harvard University, Aiken Computation Laboratory, 33 Oxford Street, Cambridge, Massachusetts 02138, October 1988.

[9] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[10] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[11] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[12] 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.

[13] W. E. Weihl. Distributed Version Management of Read-only Actions. *IEEE Transactions on Software Engineering*, 13(2):55–64, January 1987.

[14] W. E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–283, April 1989.