# Multiversion Concurrency Control—Theory and Algorithms

PHILIP A. BERNSTEIN and NATHAN GOODMAN
Harvard University

Concurrency control is the activity of synchronizing operations issued by concurrently executing programs on a shared database. The goal is to produce an execution that has the same effect as a serial (noninterleaved) one. In a multiversion database system, each write on a data item produces a new copy (or *version*) of that data item. This paper presents a theory for analyzing the correctness of concurrency control algorithms for multiversion database systems. We use the theory to analyze some new algorithms and some previously published ones.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems.

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Transaction processing

## 1. INTRODUCTION

A *database system* (DBS) is a process that executes read and write operations on data items of a database. A *transaction* is a program that issues reads and writes to a DBS. When transactions execute concurrently, the interleaved execution of their reads and writes by the DBS can produce undesirable results. *Concurrency control* is the activity of avoiding such undesirable results. Specifically, the goal of concurrency control is to produce an execution that has the same effect as a serial (noninterleaved) one. Such executions are called *serializable.*

A DBS attains a serializable execution by controlling the order in which reads and writes are executed. When an operation is submitted to the DBS, the DBS can either execute the operation immediately, delay the operation for later processing, or reject the operation. If an operation is rejected, then the transaction that issued the operation is *aborted,* meaning that all of the transaction's writes are undone, and transactions that read any of the values produced by those writes are also aborted.

The principal reason for rejecting an operation is that it arrived "too late." For

This work was supported by N.S.F. Grant MCS-79-07762, by the Office of Naval Research under Contract N00014-80-C-647, by Rome Air Development Center under Contract F30602-81-C-0028, and by Digital Equipment Corporation.

Authors' addresses: P. Bernstein, Sequoia Systems, Inc., 1 Metropolitan Corp. Center, Boston Park West, Marlborough, MA 01752; N. Goodman, Computer Science Department, Boston University, Boston, MA 02215.

example, a read is normally rejected because the value it was supposed to read has already been overwritten. Such rejections can be avoided by keeping old copies of each data item. Then a tardy read can be given an old value of a data item, even though it was "overwritten."

In a *multiversion* DBS, each write on a data item $x$, say, produces a new copy (or *version*) of $x$. For each read on $x$, the DBS selects one of the versions of $x$ to be read. Since writes do not overwrite each other, and since reads can read any version, the DBS has more flexibility in controlling the order of reads and writes. Several interesting concurrency control algorithms that exploit multiversions have been proposed [1, 2, 6, 7, 17, 19, 20, 21]. Theoretical work on this problem includes [15] and [21].

This paper presents a theory for analyzing the correctness of concurrency control algorithms for multiversions DBSs. We present some new multiversion algorithms. We use the theory to analyze the new algorithms and several previously published ones.

Section 2 reviews concurrency control theory for nonmultiversion databases. Section 3 extends the theory to multiversion databases. Sections 4–6 use the theory to analyze multiversion concurrency control algorithms.

## 2. BASIC SERIALIZABILITY THEORY

The standard theory for analyzing database concurrency control algorithms is *serializability theory* [4, 5, 8, 14, 16, 21]. Serializability theory is a method for analyzing executions allowed by the concurrency control algorithm. The theory gives a precise condition under which an execution is correct. A concurrency control algorithm is then judged to be correct if all of its executions are correct.

This section reviews serializability theory for concurrency control without multiversions.

### 2.1 System Model

We assume the DBS is distributed and use Lamport's model of distributed executions [13]. The system consists of a collection of processes that communicate by passing messages. The model describes an execution in terms of a *happens-before relation* that tells the order in which events occur. An *event* is one of the following: the execution of an operation by a process, the sending of a message, or the receipt of a message.

Within a process, the happens-before relation is any partial order over the events of the process. For the system, the happens-before relation (denoted $<$) is the smallest partial order over all events in the system such that: (1) if $e$ and $f$ are events in process $P$, and $e$ happens before $f$ in $P$, then $e < f$; (2) if $e$ is the event "process $P$ sends message $M$" and $f$ is the event "process $Q$ receives $M'$," then $e < f$. Condition (1) states that $<$ must be consistent with the order of events within each process. Condition (2) states that a message must be sent before it is received. And, since $<$ is the smallest partial order satisfying these conditions, *condition (2) is the only way that events in different processes can be ordered.*

This paper deals at a higher level of abstraction. Hereafter, we will not explicitly mention processes and messages (except briefly in Section 6). For concreteness, the reader may assume that each transaction is a process, and each data item is

managed by a separate process. (Our results do not depend on these assumptions.) Under these assumptions each database operation entails two message exchanges. For transaction $T_i$ to read $x$, $T_i$ must send a message to $x$'s process; to return $x$'s value, the $x$ process must send a message to $T_i$. The same message pattern is needed for writes; in this case, the return message just acknowledges that the write has been done. Also under these assumptions, any decision or event ordering involving one data item is a local activity; decisions or orderings involving multiple data items are distributed activities. The abstraction that we use hides message exchanges and related issues, allowing us to reason about concurrency control at a higher level.

## 2.2 Logs

Serializability theory models executions by *logs*. A log identifies the read and write operations executed on behalf of each transaction, and tells the order in which those operations were executed. A log is an abstraction of Lamport's happens-before relation.

A *transaction log* represents an allowable execution of a single transaction. Formally, a transaction log is a partially ordered set (*poset*) $T_i = (\sum_i, <_i)$ where $\sum_i$ is the set of reads and writes issued by (an execution of) transaction $i$, and $<_i$ tells the order in which those operations must be executed. We write transaction logs as diagrams.

$$T_1 = \begin{matrix} r_1[x]\searrow \\ \\ r_1[z]\nearrow \end{matrix} \quad w_1[x].$$

$T_1$ represents a transaction that reads $x$ and $z$ in parallel, and then writes $x$. (Presumably, the value written depends on the values read.)

We use $r_i[x]$ ($w_i[x]$) to denote a read (write) on $x$ issued by $T_i$. To keep this notation unambiguous, we assume that no transaction reads or writes a data item more than once. None of our results depend on this assumption.

Let $T = \{T_0, \ldots, T_n\}$ be a set of transaction logs. A *DBS log* (or simply a *log*) over $T$ represents an execution of $T_0, \ldots, T_n$. Formally, a log over $T$ is a poset $L = (\sum, <)$, where

(1) $\sum = \cup_{i=0}^{n} \sum_i$;
(2) $< \supseteq \cup_{i=0}^{n} <_i$;
(3) every $r_j[x]$ is preceded by at least one $w_i[x]$ ($i = j$ is possible), where $w_i[x]$ *precedes* $r_j[x]$ is synonymous with $w_i[x] < r_j[x]$; and
(4) all pairs of conflicting operations are $<$ related (two operations *conflict* if they operate on the same data item, and at least one is a write).

Condition (1) states that the DBS executes all and only those operations submitted by $T_0, \ldots, T_n$. Condition (2) states that the DBS honors all operation orderings stipulated by the transactions. Condition (3) states that no transaction can read a data item until some transaction has written its initial value. Condition (4) states that the DBS executes conflicting operations sequentially. For example, if $T_i$ reads $x$ and $T_j$ writes $x$, $r_i[x]$ happens before $w_j[x]$ or vice versa; the operations cannot occur at the same time.

Consider the following transaction logs:

$$T_0 = \begin{matrix} w_0[x] \\ w_0[y], \\ w_0[z] \end{matrix} \qquad T_1 = \begin{matrix} r_1[x] & & \\ & \searrow & \\ & & w_1[x]. \\ & \nearrow & \\ r_1[z] & & \end{matrix}$$

The following are some of the possible logs over $\{T_0, T_1\}$:

$$\begin{matrix} w_0[x] \to r_1[x] & & \\ & \searrow & \\ w_0[y] \quad \downarrow & & w_1[x], \\ & \nearrow & \\ w_0[z] \to r_1[z] & & \end{matrix} \tag{1}$$

$$\begin{matrix} w_0[x] \to r_1[x] & & \\ & \searrow & \\ w_0[y] \quad \uparrow & & w_1[x], \\ & \nearrow & \\ w_0[z] \to r_1[z] & & \end{matrix} \tag{2}$$

$$\begin{matrix} w_0[x] \to r_1[x] & & \\ & \searrow & \\ w_0[y] & & w_1[x]. \\ & \nearrow & \\ w_0[z] \to r_1[z] & & \end{matrix} \tag{3}$$

Note that orderings implied by transitivity are usually not drawn. For example, $w_0[x] < w_1[x]$ is not drawn in the diagrams, although it follows from $w_0[x] < r_1[x] < w_1[x]$.

Notice that the DBS is allowed to process read($x$) and read($z$) sequentially (cf. (1) and (2)), even though $T_1$ allows them to run in parallel. However, the DBS is not allowed to reverse or eliminate any ordering stipulated by $T_1$.

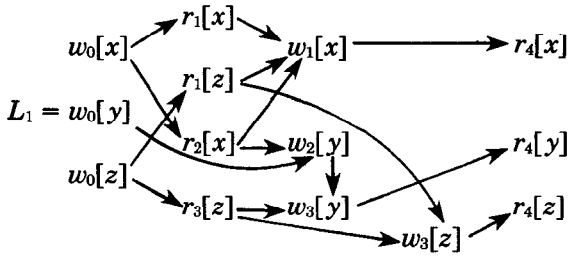
Given transaction logs

$$T_0 = \begin{matrix} w_0[x] \\ w_0[y] \\ w_0[z], \end{matrix} \qquad T_1 = \begin{matrix} r_1[x] & & \\ & \searrow & \\ & & w_1[x], \\ & \nearrow & \\ r_1[z] & & \end{matrix}$$

$$T_2 = r_2[x] \to w_2[y], \qquad T_3 = r_3[z] \begin{matrix} \nearrow w_3[y] \\ \searrow w_3[z] \end{matrix},$$

$$T_4 = \begin{matrix} r_4[x] \\ r_4[y], \\ r_4[z]. \end{matrix}$$

the following is a log over $\{T_0, T_1, T_2, T_3, T_4\}$:




The following is another log over the same transactions:

$$L_2 = w_0[x]w_0[y]w_0[z]r_2[x]w_2[y]r_1[x]r_1[z]w_1[x]r_3[z]w_3[y]w_3[z]r_4[x]r_4[y]r_4[z].$$

When we write a log as a sequence, for example, $L_2$, we mean that the log is totally ordered: Each operation precedes the next one and all subsequent ones in the sequence. Thus, in $L_2$, $w_0[x] < w_0[y] < w_0[z] < r_2[x] \ldots$.

## 2.3 Log Equivalence

Intuitively, two logs are equivalent if each transaction performs the same computation in both logs. We formalize log equivalence in terms of information flow between transactions.

Let $L$ be a log over $\{T_0, \ldots, T_n\}$. Transaction $T_j$ *reads-x-from* $T_i$ in $L$ if (1) $w_i[x]$ and $r_j[x]$ are operations in $L$; (2) $w_i[x] < r_j[x]$; and (3) no $w_k[x]$ falls between these operations. Two logs over $\{T_0, \ldots, T_n\}$ are *equivalent*, denoted $\equiv$, if they have the same reads-from relationships; that is, for all $i$, $j$, and $x$, $T_j$ reads-$x$-from $T_i$ in one log iff this condition holds in the other. This definition ensures that each transaction reads the same values from the database in both logs.

Consider logs $L_1$ and $L_2$ of the previous section. These logs have the same read-from's:

$T_1$ reads-$x$-from $T_0$, $T_1$ reads-$z$-from $T_0$;
$T_2$ reads-$x$-from $T_0$;
$T_3$ reads-$z$-from $T_0$;
$T_4$ reads-$x$-from $T_1$, $T_4$ reads-$y$-from $T_3$,
$T_4$ reads-$z$-from $T_3$.

Therefore, $L_1 \equiv L_2$.

This definition of log equivalence ignores the final database state produced by the logs. For example, the logs

$$L = w_0[x]w_1[x] \quad \text{and} \quad L' = w_1[x]w_0[x]$$

are equivalent, even though different transactions produce the final value of $x$ in each log. It is often desirable to strengthen the notion of equivalence by insisting that for each $x$, the same transaction writes the final value of $x$ in both logs. This can be modeled by (1) adding a "final transaction" that follows all other transactions and reads the entire database (e.g., $T_4$ in logs $L_1$ and $L_2$); and (2) redefining

equivalence to be that the logs have the same reads-from's and the same final transaction.

### 2.4 Serializable Logs

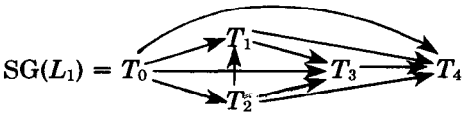
A *serial log* is a totally ordered log on $\sum$ such that for every pair of transactions $T_i$ and $T_j$, either all the operations of $T_i$ precede all those of $T_j$, or vice versa (e.g., $L_2$ in Section 2.2). A serial log represents an execution in which there is no concurrency whatsoever; each transaction executes from beginning to end before the next transaction begins. From the point of view of concurrency control, therefore, every serial log represents an obviously correct execution.

What other logs represent correct executions? From the point of view of concurrency control, a correct execution is one in which concurrency is invisible. That is, an execution is correct if it is equivalent to an execution in which there is no concurrency. Serial logs represent the latter executions, and so a *correct log* is any log equivalent to a serial log. Such logs are termed *serializable* (SR).

Log $L_1$ of Section 2.2 is SR, because it is equivalent to serial log $L_2$ of Section 2.3. Therefore $L_1$ is a correct log.

### 2.5 The Serializability Theorem

Let $L$ be a log over $\{T_0, \ldots, T_n\}$. The *serialization graph* for $L$, SG($L$), is a directed graph whose nodes are $T_0, \ldots, T_n$, and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that some operation of $T_i$ precedes and conflicts with some operation of $T_j$. The serialization graph of log $L_1$, for example, is




Edge $T_0 \rightarrow T_1$ is present because $w_0[x] < r_1[x]$, edge $T_1 \rightarrow T_3$ is present because $r_1[z] < w_3[z]$, and so forth.

SERIALIZABILITY THEOREM [4, 8, 14, 16, 21]. *If SG($L$) is acyclic, then $L$ is SR.*

## 3. MULTIVERSION SERIALIZABILITY THEORY

In a multiversion DBS, each write produces a new version. We denote versions of $x$ by $x_i$, $x_j$, ..., where the subscript is the index of the transaction that wrote the version. Operations on versions are denoted $r_i[x_j]$ and $w_i[x_i]$.

### 3.1 Multiversion Logs

Let $T = \{T_0, \ldots, T_n\}$ be a set of transaction logs (defined exactly as in Section 2.2, i.e., the operations reference data items). To execute $T$, a multiversion DBS must translate $T$'s "data item operations" into "version operations." We formalize this translation by a function $h$ which maps each $w_i[x]$ into $w_i[x_i]$, and each $r_i[x]$ into $r_i[x_j]$ for some $j$.

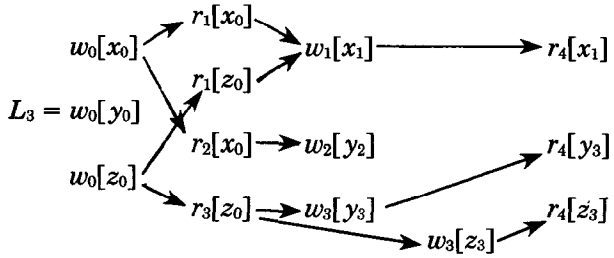
A *multiversion DBS log* (or simply *MV log*) over $T$ is a poset $L = (\sum, <)$ where

(1) $\sum = h(\cup_{i=1}^{n} \sum_i)$ for some translation function $h$,
(2) for each $T_i$ and all operations $\text{op}_i$ and $\text{op}_i'$, if $\text{op}_i <_i \text{op}_i'$ then $h(\text{op}_i) < h(\text{op}_i')$, and

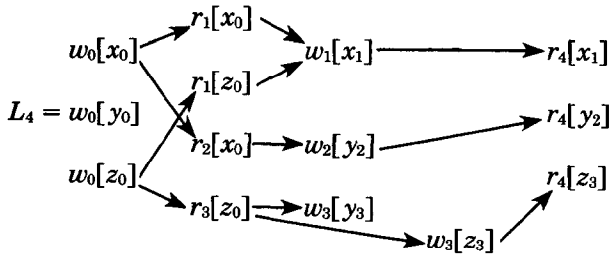
(3) if $h(r_j[x]) = r_j[x_i]$, then $w_i[x_i] < r_j[x_i]$.

Condition (1) states that each operation submitted by a transaction is translated into an appropriate multiversion operation. Condition (2) states that the MV log preserves all orderings stipulated by transactions. Condition (3) states that a transaction may not read a version until it has been produced.

The following is an MV log over $\{T_0, T_1, T_2, T_3, T_4\}$ of Section 2.




All MV logs over a set $T$ have the same write operations, since $h(w_i[x]) = w_i[x_i]$, but they need not have the same reads. For example, $L_4$ has $r_4[y_2]$ instead of $r_4[y_3]$.




### 3.2 MV Log Equivalence

Most definitions and results from basic serializability theory extend to MV logs; we simply replace the notion of "data item" by "version" in those definitions and results. However, the structure of MV logs simplifies the treatment. This section restates the material of Sections 2.3 and 2.4 for MV logs.

Let $L$ be an MV log over $\{T_0, \ldots, T_n\}$. Transaction $T_j$ reads-$x$-from $T_i$ in $L$ if $T_j$ reads the version of $x$ produced by $T_i$. By definition, the version of $x$ produced by $T_i$ is $x_i$. So, $T_j$ reads-$x$-from $T_i$ iff $T_j$ reads $x_i$. This means that *the reads-from relationships in L are determined by the translation function h*, namely, by the way $h$ translates "data item reads" into "version reads."

Two MV logs over $\{T_0, \ldots, T_n\}$ are equivalent, denoted $\equiv$, if they have the same reads-from relationships. The reads-from relationships in an MV log are determined by its read operations: $T_j$ reads-$x$-from $T_i$ iff $r_j[x_i]$ is an operation of the log. So, two logs are equivalent iff they have the same read operations. Moreover, since all MV logs over the same transactions have the same writes, equivalence reduces to a trivial condition.

FACT 1. *Two MV logs over a set of transactions T are equivalent iff the logs have the same operations.*

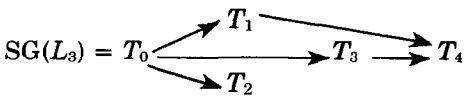
Two "version operations" *conflict* if they operate on the same version and one is a write. Only one pattern of conflict is possible in an MV log: If $op_i < op_j'$ and these operations conflict, then $op_i$ is $w_i[x_i]$ and $op_j$ is $r_j[x_i]$. Conflicts of the form $w_i[x_i] < w_j[x_i]$ are impossible, because each write produces a new version. Conflicts of the form $r_j[x_i] < w_i[x_i]$ are impossible since $T_j$ cannot read $x_i$ until it has been produced. Thus all conflicts in an MV log correspond to reads-from relationships.

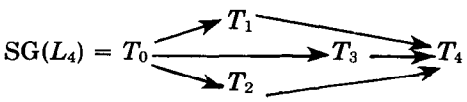
The *serialization graph* for an MV log is defined as for a regular log. Since conflicts are highly structured in an MV log, serialization graphs are quite simple. Let $L$ be an MV log over $\{T_0, \ldots, T_n\}$. SG($L$) has nodes $T_0, \ldots, T_n$ and edges $T_i \rightarrow T_j$ ($i \neq j$) such that for some $x$, $T_j$ reads-$x$-from $T_i$. That is, $T_i \rightarrow T_j$ is present iff some $x$, $r_j[x_i]$ is an operation of $L$. This gives us the following.

FACT 2. *Let $L$ and $L'$ be MV logs over $T$.*

(1) *If $L$ and $L'$ have the same operations, then $SG(L) = SG(L')$.*
(2) *If $L$ and $L'$ are equivalent, then $SG(L) = SG(L')$.*

The serialization graphs for logs $L_3$ and $L_4$ of the previous section are given below:




(Cf. SG($L_1$) in Section 2.4.)

### 3.3 One-Copy Serializability

Although the database has multiple versions, users expect their transactions to behave as if there were just one copy of each data item. Serial logs do not always behave this way. Here is a simple example.

$$w_0[x_0]w_0[y_0]r_1[x_0]w_1[y_1]r_2[y_0]w_2[x_2].$$

$T_2$ reads-$y$-from $T_0$ even though $T_1$ comes between $T_0$ and $T_2$ and produces a new value for $y$. This behavior cannot be reproduced with only one copy of $y$. In a one-copy database, if $T_0$ comes before $T_1$ and $T_1$ is before $T_2$, then $T_2$ must read the value of $y$ produced by $T_1$.

We must therefore restrict the set of allowable serial logs.

A serial MV log $L$ is *one-copy serial* (or *1-serial*) if for all $i, j$, and $x$, if $T_j$ reads-$x$-from $T_i$ then $i = j$ or $T_i$ is the last transaction preceding $T_j$ that writes into *any* version of $x$. (Since $L$ is totally ordered, the word "last" in this definition is well-defined.) The log above is not 1-serial, because $T_2$ reads-$y$-from $T_0$, but $w_0[y_0] < w_1[y_1] < r_2[y_0]$. $L_5$ below is 1-serial.

$$L_5 = w_0[x_0]w_0[y_0]w_0[z_0]r_2[x_0]w_2[y_2]r_1[x_0]r_1[z_0]w_1[x_1]r_3[z_0]w_3[y_3]w_3[z_3]r_4[x_1]r_4[y_3]r_4[z_3].$$

A log is *one-copy serializable* (or *1-SR*) if it is equivalent to a 1-serial log. For example, $L_3$ of Section 3.1 is equivalent to $L_5$, as can be verified by Fact 1; hence $L_3$ is 1-SR. $L_4$ is equivalent to no 1-serial log (this can be verified by checking all possible serial logs with the same operations as $L_4$); hence $L_4$ is not 1-SR.

It is possible for a serial log to be 1-SR even though it is not 1-serial itself. For example,

$$w_0[x_0]r_1[x_0]w_1[x_1]r_2[x_0]$$

is not 1-serial since $T_2$ reads-$x$-from $T_0$ instead of $T_1$. But it is 1-SR, because it is equivalent to

$$w_0[x_0]r_2[x_0]r_1[x_0]w_1[x_1].$$

One-copy serializability is our correctness criterion for multiversion concurrency control. The following theorem justifies this criterion, proving that an MV log behaves like a serial non-MV log iff the MV log is 1-SR.

First, we extend our notion of log equivalence to handle MV and non-MV logs. Let $L$ and $L'$ be (MV or non-MV) logs over $T$. $L$ and $L'$ are *equivalent*, $\equiv$, if they have the same reads-from relationships.

1-SR EQUIVALENCE THEOREM. *Let $L$ be an MV log over $T$. $L$ is equivalent to a serial, non-MV log over $T$ iff $L$ is* 1-SR.

PROOF.

(If). Let $L_s$ be a 1-serial log equivalent to $L$. Form a serial, non-MV log $L'_s$ by translating each $w_i[x_i]$ into $w_i[x]$ and $r_j[x_i]$ into $r_j[x]$. Consider any reads-from relationship in $L_s$, say $T_j$ reads-$x$-from $T_i$. Since $L_s$ is 1-serial, no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$. Hence no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$ in $L'_s$. Thus, $T_j$ reads-$x$-from $T_i$ in $L'_s$. This establishes $L'_s \equiv L_s$. Since $L_s \equiv L$, $L \equiv L'_s$ follows by transitivity (since $\equiv$ is an equivalence relation).

(Only if). Let $L'_s$ be the hypothesized serial, non-MV log equivalent to $L$. Translate $L'_s$ into a serial MV log $L_s$ by mapping each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ such that $T_j$ reads-$x$-from $T_i$ in $L'_s$. This translation preserves reads-from relationships, so $L_s \equiv L'_s$. By transitivity, $L \equiv L_s$.

It remains to prove that $L_s$ is 1-serial. Consider any reads-from relationship in $L'_s$, say $T_j$ reads-$x$-from $T_i$. Since $L'_s$ is a non-MV log, no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$. Hence no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$ in $L_s$. Thus, $L_s$ is 1-serial, as desired. □

### 3.4 The 1-Serializability Theorem

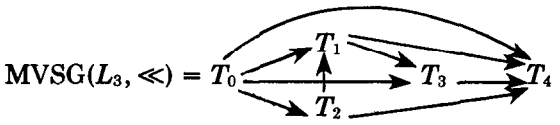
To tell if an MV log is 1-SR we use a modified serialization graph. Given a log $L$ and data item $x$, a *version order* for $x$ is any (nonreflexive) total order over all of the versions of $x$ written in $L$. A *version order*, $\ll$, for $L$ is the union of the version orders for all data items. A possible version order for $L_3$ of Section 3.1 (or $L_5$ of Section 3.3) is

$$\ll = \begin{cases} x_0 \ll x_1 \\ y_0 \ll y_2 \ll y_3. \\ z_0 \ll z_3 \end{cases}$$

Given $L$ and a version order $\ll$, the *multiversion serialization graph*, MVSG($L$, $\ll$), is SG($L$) with the following edges added:

(1) for each $r_k[x_j]$ and $w_i[x_i]$ in $L$, $k \neq i$, if $x_i \ll x_j$ then include $T_i \rightarrow T_j$, else include $T_k \rightarrow T_i$.

For example,




(Cf. SG($L_1$) in Section 2.5.)

The following theorem is our principal tool for analyzing multiversion concurrency control algorithms.

1-SERIALIZABILITY THEOREM. *An MV log $L$ is 1-SR iff there exists a version order $\ll$ such that MVSG($L$, $\ll$) is acyclic.*

PROOF

(If). Let $L_s$ be a serial MV log induced by a topological sort of MVSG($L$, $\ll$). That is, $L_s$ is formed by topologically sorting MVSG($L$, $\ll$), and as each node $T_i$ is listed in the sort, the operations of $T_i$ in $L$ are added to $L_s$ one by one in any order consistent with $L$. $L_s$ has the same operations as $L$, so by Fact 1, $L \equiv L_s$.

It remains to prove that $L_s$ is 1-serial. Consider any reads-from situation, say, $T_k$ reads-$x$-from $T_j$. Let $w_i[x_i]$ be any other write on a version of $x$. If $x_i \ll x_j$, then by rule (1) of the MVSG definition, the graph includes $T_i \rightarrow T_j$. This edge forces $T_j$ to follow $T_i$ in $L_s$. If $x_j \ll x_i$, then by rule (1), MVSG($L$, $\ll$) includes $T_k \rightarrow T_i$. This forces $T_k$ to precede $T_i$ in $L_s$. In both cases, $T_i$ is prevented from falling between $T_j$ and $T_k$. Since $T_i$ was an arbitrary writer on $x$, this proves that no transaction that writes a version of $x$ comes between $T_j$ and $T_k$ in $L_s$. Thus $L_s$ is 1-serial.

(Only if). Given $L$ and $\ll$, let MV($L$, $\ll$) be the graph specified by statement (1) of the MVSG definition. Statement (1) depends only on the operations in $L$ and $\ll$; it does *not* depend on the *order* of operations in $L$. Thus, if $L_1$ and $L_2$ are multiversion logs with the same operations, then MV($L_1$, $\ll$) = MV($L_2$, $\ll$), for all version orders $\ll$.

Let $L_s$ be a 1-serial log equivalent to $L$. All edges in SG($L_s$) go "left-to-right", that is, if $T_i \rightarrow T_j$, then $T_i$ is before $T_j$ in $L_s$. Define $\ll$ as follows: $x_i \ll x_j$ only if $T_i$ is before $T_j$ in $L_s$. All edges in MV($L_s$, $\ll$) are also left-to-right. Therefore all edges in MVSG($L_s$, $\ll$) = MV($L_s$, $\ll$) $\cup$ SG($L_s$) are left-to-right, too. This implies that MVSG($L_s$, $\ll$) is acyclic.

By Fact 1, $L$ and $L_s$ have the same operations. Hence, MV($L$, $\ll$) = MV($L_s$, $\ll$). By Fact 2, SG($L$) = SG($L_s$). Therefore MVSG($L$, $\ll$) = MVSG($L_s$, $\ll$). Since MVSG($L_s$, $\ll$) is acyclic, so is MVSG($L$, $\ll$). □

Sections 4–6 use the 1-Serializability Theorem to analyze multiversion concurrency control algorithms. We conclude this section with a complexity result.

### 3.5 1-Serializability Is NP-Complete

1-SR COMPLEXITY THEOREM. *It is NP-complete to decide whether an MV log is 1-SR.*

PROOF

(Membership in NP). Let $L$ be an MV log over $T$. Guess a 1-serial log $L_s$ over $T$ and verify $L_s \equiv L$. By Fact 1, we can verify $L_s \equiv L$ by comparing the logs' operation sets.

(NP-hardness). The reduction is from the log SR problem (Problem SR 33 in [9, 14, 16]). Let $L'$ be a non-MV log over $T$. Map $L'$ into an equivalent MV log $L$ by translating each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ such that $T_j$ reads-$x$-from $T_i$ in $L'$. By the 1-SR Equivalence Theorem, $L$ is 1-SR iff there exists a non-MV serial log $L_s'$ such that $L \equiv L_s'$. But, by transitivity, $L_s'$ exists iff $L'$ is SR. Thus $L'$ is SR iff $L$ is 1-SR. □

Papadimitriou and Kanellakis prove that a related problem is NP-complete [15]: Given a conventional log $L$, can one transform $L$ into a 1-SR MV log by mapping each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ for some $x_i$ where $w_i[x] < r_j[x]$? This problem corresponds to choosing versions for reading after having scheduled the operations. Our problem corresponds to choosing versions at the same time as scheduling the operations.

## 4. MULTIVERSION TIMESTAMPING

The earliest multiversion concurrency control algorithm that we know of is Reed's multiversion timestamping algorithm [17].

Each transaction, $T_i$, is assigned a unique *timestamp*, TS($i$), when it begins executing. Intuitively, the timestamp tells the "time" at which the transaction began. Formally, timestamps are just numbers with the property that each transaction is assigned a different timestamp. Each read and write carries the timestamp of the transaction that issued it, and each version carries the timestamp of the transaction that wrote it.

Operations are processed first-come-first-served. But the translation from data item operations to version operations makes it appear as if operations were processed in timestamp order.

The algorithm works as follows.

(1) $r_i[x]$ is translated into $r_i[x_k]$, where $x_k$ is the version of $x$ with largest timestamp $\leq$ TS($i$).
(2) $w_i[x]$ has two cases. If the DBS has already processed $r_j[x_k]$ such that TS($k$) < TS($i$) < TS($j$), then $w_i[x]$ is *rejected.* Otherwise $w_i[x]$ is translated into $w_i[x_i]$. Intuitively, $w_i[x]$ is rejected if it would invalidate $r_j[x_k]$.

We wish to use serializability theory to prove this algorithm correct. To do so, we must state the algorithm in terms of serializability theory. We take the description of the algorithm above and infer properties that all logs produced by the algorithm will satisfy. *These properties form our formal definition of the algorithm.* We use serializability theory to prove that these log properties imply 1-serializability.

The following properties form our formal definition of the *MV timestamping algorithm.* Let $L$ be an MV log over $\{T_0, \ldots, T_n\}$.

TS1. Every $T_i$ has a numeric timestamp TS($i$) satisfying a uniqueness condition: TS($i$) = TS($j$) iff $i = j$.
TS2. All $r_k[x_j]$ and $w_i[x_i]$ are <-related; that is, $r_k[x_j] < w_i[x_i]$, or vice versa.

TS3.1. For every $r_k[x_j]$, $TS(j) \leq TS(k)$.

TS3.2. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $w_i[x_i] < r_k[x_j]$, then either $TS(i) < TS(j)$ or $TS(k) \leq TS(i)$.

TS4. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $r_k[x_j] < w_i[x_i]$, then either $TS(i) < TS(j)$ or $TS(k) \leq TS(i)$.

Property TS1 just says that transactions have unique timestamps. TS2 is implicit in the description of how the algorithm works; without this property, the condition, "If the DBS has already processed $r_j[x_k]$ . . ." is not well-defined. TS3 states that at the time $r_k[x_j]$ is processed, $x_j$ is the version of $x$ with the largest timestamp $\leq TS(k)$. TS4 states that once the DBS has processed $r_k[x_j]$, it will not process any $w_i[x_i]$ with $TS(j) \leq TS(i) < TS(k)$.

Properties TS3.2 and TS4 can be simplified. By TS2, $r_k[x_j]$ and $w_i[x_i]$ are $<$-related. So TS3.2 and TS4 are equivalent to the following.

TS5. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, either $TS(i) < TS(j)$ or $TS(k) \leq TS(i)$.

We now prove that any log satisfying these properties is 1-SR. In other words, MV timestamping is a correct concurrency control algorithm.

MULTIVERSION TIMESTAMPING THEOREM. *All logs produced by the MV timestamping algorithm are 1-SR.*

PROOF. Let $L$ be a log produced by the algorithm. Define a version order as follows: $x_i \ll x_j$ implies $TS(i) < TS(j)$. We prove that all edges in $MVSG(L, \ll)$ are in timestamp order: If $T_i \rightarrow T_j$ is an edge, then $TS(i) < TS(j)$.

Let $T_i \rightarrow T_j$ be an edge of $SG(L)$. This edge corresponds to a reads-from situation, that is, for some $x$, $T_j$ reads-$x$-from $T_i$. By TS3.1, $TS(i) \leq TS(j)$; by TS1, $TS(i) \neq TS(j)$. So $TS(i) < TS(j)$, as desired.

Consider any edge introduced by rule (1) of the MVSG definition. Let $w_i[x_i]$, $w_j[x_j]$, and $r_k[x_j]$ be the operations stipulated by rule (1). There are two cases.

(1) $x_i \ll x_j$.

Then the edge is $T_i \rightarrow T_j$. $TS(i) < TS(j)$ comes from our definition of $\ll$.

(2) $x_j \ll x_i$.

Then the edge is $T_k \rightarrow T_i$. By TS5, either $TS(i) < TS(j)$ or $TS(k) \leq TS(i)$. The first option is impossible, since the definition of $\ll$ requires $TS(j) < TS(i)$. By TS1, $TS(k) \neq TS(i)$. So, $TS(k) < TS(i)$, as desired.

This proves that all edges in $MVSG(L, \ll)$ are in timestamp order. Since timestamps are numbers, hence totally ordered, it follows that $MVSG(L, \ll)$ is acyclic. So by the 1-serializability theorem, $L$ is 1-SR. □

## 5. MULTIVERSION LOCKING

Bayer et al. [1, 2] and Stearns and Rosenkrantz [20] have presented multiversion algorithms that synchronize using a technique similar to locking. This section studies a generalization of their algorithms. As in the previous section, we start with an informal description of the algorithm. Then we state log properties induced by the algorithm. Finally we prove that these log properties imply 1-serializability.

Each transaction and version exists in one of two states: *certified* or *uncertified.* When a transaction begins, it is uncertified; when a version is written, it, too, is uncertified. Later actions of the algorithm cause the transaction and all versions it wrote to become certified. The concept of "certified" corresponds to "closed" in [20].

Let $c_i[x_i]$ be the event "$x_i$ is certified." The algorithm requires that all $c_i[x_i]$ and $r_k[x_j]$ be $<$-related. Also, all $c_i[x_i]$ and $c_j[x_j]$ must be $<$-related. A version order is defined thus: $x_i \ll x_j$ iff $c_i[x_i] < c_j[x_j]$.

The algorithm works as follows.

First, $r_i[x]$ is translated into $r_i[x_k]$, where $x_k$ is either the *last* (with respect to $\ll$) certified version of $x$ or *any* uncertified version. The algorithm may use any rule whatever for deciding which of these versions to read.

Then, $w_i[x]$ is translated into $w_i[x_i]$. As stated above, $x_i$ is uncertified at this point.

Finally, when a transaction finishes executing, the DBS attempts to certify it and all versions it wrote. For each data item $x$ that $T_i$ wrote, the DBS tries to set a *certify-lock* on $x$ for $T_i$. This succeeds iff no other transaction already has a certify-lock on $x$; if the lock cannot be set, $T_i$ waits until it can. When $T_i$ has all of its certify-locks, two further conditions must be satisfied:

C1. For each $x_k$ that $T_i$ read, $k \neq i$, $x_k$ is certified.
C2. For each $x_i$ that $T_i$ wrote, and for each version $x_k$ of $x$ that is already certified, all transactions that read $x_k$ have been certified.

Attaining C1 is just a matter of time; once C1 is satisfied no future event can cause it to become false. To attain C2, we set a *certify-token* on $x$ to stop future reads from reading certified versions of $x$; instead, they may read $x_i$ or any other uncertified version of $x$.

When these conditions hold, $T_i$ is declared to be certified. This fact is broadcast to all versions $T_i$ wrote. When a version $x_i$ receives this information, it, too, is certified, that is, the event $c_i[x_i]$ occurs. When $x_i$ is certified, the certify-lock and certify-token on $x_i$ are released.

This algorithm, like most locking algorithms, can deadlock. Deadlocks can arise from two independent causes: waiting for certify-locks, and waiting for conditions C1 and C2. To detect deadlocks, the algorithm can use a directed *blocking graph* whose nodes are the transactions, and whose edges are all $T_i \rightarrow T_j$ such that $T_i$ is blocking the progress of $T_j$. There is a deadlock iff the graph has a cycle [11, 12]. Deadlock prevention schemes such as those in [3, 18] can also be used. The system should keep track of the two types of deadlock separately. To resolve deadlocks caused by certify-locks, the system should force one or more transactions to give up enough of their certify-locks to break the deadlock; these transactions can try later to get these locks back. To break deadlocks caused by C1 and C2, the system must *abort* one or more transactions. (Cascading abort is possible if the algorithm allows transactions to read uncertified versions.)

The algorithm induces the following log properties. These properties form our formal definition of the *MV locking algorithm.* Let $L$ be an MV log over $\{T_0, \ldots, T_n\}$. Let us augment $L$ with symbols that represent important events in the algorithm, specifically: for each $T_i$, let $c_i$ represent the event "$T_i$ is declared to be

certified"; for each version $x_i$ written by $T_i$, let $cl_i[x_i]$ represent "the DBS sets a certify-lock on $x$ for $T_i$"; and for each $x_i$, let $c_i[x_i]$ represent "$x_i$ is certified."

L1.1. For every $T_i$, $c_i$ follows all of the reads and writes of $T_i$.
L1.2. For every every $x_i$ written by $T_i$, $cl_i[x_i] < c_i < c_i[x_i]$.

Property L1 says that a transaction is certified after it executes; all certify-locks must be obtained before the transaction is certified; and the transaction must be certified before its versions are certified.

L2.1. Every $cl_i[x_i]$ and $cl_j[x_j]$ are $<$-related.
L2.2. For every $x_i$ and $x_j$, if $cl_i[x_i] < cl_j[x_j]$ then $c_i[x_i] < cl_j[x_j]$.

L2 says that certify-locks *conflict*—two transactions cannot simultaneously hold certify-locks on the same data item.

L3.1. Every $r_k[x_j]$ and $c_i[x_i]$ are $<$-related.
L3.2. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $c_i[x_i] < r_k[x_j]$ and $c_j[x_j] < r_k[x_j]$, then $c_i[x_i] < c_j[x_j]$.

L3 expresses the rule for translating reads. If $x_j$ is already certified at the time $r_k[x_j]$ occurs, then $x_j$ is the *last* certified version at that time.

L4.1. For every $r_k[x_j]$, $k \neq j$, $c_j[x_j] < c_k$.
L4.2. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $r_k[x_j] < c_i[x_i]$ and $c_j[x_j] < c_i$, then $c_k < c_i$.

These last properties are certification conditions C1 and C2, respectively.

The following lemmas extract useful properties from L1–L4.

LEMMA 1. *Let $T_i$ and $T_j$ be transactions that write $x$. Then*

$$\begin{array}{ll} \textit{either} & cl_i[x_i] < c_i < c_i[x_i] < cl_j[x_j] < c_j < c_j[x_j] \\ \textit{or} & cl_j[x_j] < c_j < c_j[x_j] < cl_i[x_i] < c_i < c_i[x_i]. \end{array}$$

PROOF. L2.1 requires that $cl_i[x_i]$ and $cl_j[x_j]$ be $<$-related. Suppose $cl_i[x_i] < cl_j[x_j]$. By L1.2, $cl_i[x_i] < c_i < c_i[x_i]$; by L2.2, $c_i[x_i] < cl_j[x_j]$; by L1.2 again, $cl_j[x_j] < c_j < c_j[x_j]$. This establishes the first possibility permitted by Lemma 1. If $cl_j[x_j] < cl_i[x_i]$, the same argument establishes the second possibility. □

LEMMA 2. *Properties L1–L4 imply*

*L5. For every $r_k[x_j]$, $k \neq j$, $c_j < c_k$.*
*L6. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, either $c_i < c_j$ or $c_k < c_i$.*

PROOF (L5). By L1, $c_j < c_j[x_j]$. By L4.1, $c_j[x_j] < c_k$. L5 follows by transitivity.
(L6). Using logical manipulation we can express L3.2 as

$$\text{L3.2}'.\ (c_i[x_i] < r_k[x_j]) \Rightarrow (c_i[x_i] < r_k[x_j]) \wedge \neg (c_j[x_j] < r_k[x_j]) \vee (c_i[x_i] < c_j[x_j]).$$

By L3.1, the first line on the right-hand side simplifies to

$$(c_i[x_i] < r_k[x_j]) \wedge (r_k[x_j] < c_j[x_j]).$$

By transitivity, this implies $(c_i[x_i] < c_j[x_j])$, and so the entire right-hand side

implies $c_i[x_i] < c_j[x_j]$. By Lemma 1, this implies $c_i < c_j$. So L3.2′ implies

L3.2″. $(c_i[x_i] < r_k[x_j]) \Rightarrow c_i < c_j$.

Similarly, we can express L4.2 as

L4.2′. $(r_k[x_j] < c_i[x_i]) \Rightarrow \neg(c_j[x_j] < c_i) \vee (c_k < c_i)$.

By Lemma 1, $c_j[x_j]$ and $c_i$ are <-related. So the first term on the right-hand side simplifies to $(c_i < c_j[x_j])$. By Lemma 1, again, this is equivalent to $c_i < c_j$. So L4.2′ is equivalent to

L4.2″. $(r_k[x_j] < c_i[x_i]) \Rightarrow c_i < c_j \vee c_k < c_i$.

L3.1 requires that $r_k[x_j]$ and $c_i[x_i]$ be <-related. This lets us drop the left-hand sides of L3.2″ and L4.2″, combining them into the following:

For every $r_k[x_j]$ and $c_i[x_i]$, $c_i < c_j \vee c_k < c_i$.

Since $c_i[x_i]$ exists iff $w_i[x_i]$ exists, L6 follows. □

We now prove that any log satisfying these properties is 1-SR. In other words, MV locking is a correct concurrency control algorithm.

MULTIVERSION LOCKING THEOREM. *All logs produced by the MV locking algorithm are* 1-*SR.*

PROOF. Let $L$ be a log produced by the algorithm. Define a version order as follows: $x_i \ll x_j$ implies $c_i < c_j$. We prove that all edges in MVSG($L$, $\ll$) are in certification order: If $T_i \rightarrow T_j$ is an edge, then $c_i < c_j$.

Let $T_i \rightarrow T_j$ be an edge of SG($L$). This edge corresponds to a reads-from situation, that is, for some $x$, $T_j$ reads-$x$-from $T_i$. By L5, $c_i < c_j$.

Consider any edge introduced by rule (1) of the MVSG definition. Let $w_i[x_i]$, $w_j[x_j]$, and $r_k[x_j]$ be the operations stipulated by rule (1). There are two cases.

(1) $x_i \ll x_j$: Then the edge is $T_i \rightarrow T_j$; $c_i < c_j$ comes from our definition of $\ll$.
(2) $x_j \ll x_i$: Then the edge is $T_k \rightarrow T_i$.

By L6, either $c_i < c_j$ or $c_k < c_i$. The first option is impossible, since the definition of $\ll$ requires $c_j < c_i$. So, $c_k < c_i$ as desired.

This proves that all edges in MVSG($L$, $\ll$) are in certification order. Since the certification order is embedded in a partial order (namely $L$), it follows that MVSG($L$, $\ll$) is acyclic. So, by the 1-Serializability Theorem, $L$ is 1-SR. □

The Stearns and Rosenkrantz algorithm [20] differs from ours in two respects. Theirs allows at most one uncertified version of a data item to exist at any point in time, by requiring that write operations set write-locks. Consequently, their algorithm never needs more than two versions of any data item: one certified version and at most one uncertified version. This fits nicely with database recovery [10]. Stearns and Rosenkrantz identify the certified version of a data item with its "before-value," and the uncertified version with its "after-value." The other difference involves deadlock handling. Their algorithm uses an interesting new deadlock avoidance scheme based on timestamps.

The Bayer et al. algorithm [1, 2] also uses at most two versions of each data item. As in [20], the versions of a data item are identified with its before- and after-values. Unlike Stearns and Rosenkrantz, Bayer et al. use the blocking graph to help translate data item reads into version reads. They prove that they can always select a correct version to read. That is, reads never cause a log to become non-1-SR and never cause deadlocks. This is a good property since it allows read-only transactions (*queries*) to run with little synchronization delay and no danger of deadlock.

## 6. MULTIVERSION MIXED METHOD

Prime Computer, Inc., has developed an interesting multiversion algorithm [7]. Prime's algorithm, like those at the end of Section 5, integrates concurrency control with database recovery. Unlike those algorithms, Prime's algorithm can exploit multiple certified versions of data items. Computer Corporation of America has adopted Prime's algorithm for its Adaplex DBS [6]. This section studies a generalization of Prime's algorithm.

The algorithm we study is called a mixed method. A *mixed method* is a concurrency control algorithm that combines locking with timestamping [3]. Mixed methods introduce a new problem: *consistent timestamp generation.* A timestamping algorithm uses timestamps to order conflicting transactions; intuitively, if $T_i$ and $T_j$ conflict, then $T_i$ is synchronized before $T_j$ iff $\mathrm{TS}(i) < \mathrm{TS}(j)$. A locking algorithm orders transactions on-the-fly; intuitively, if $T_i$ and $T_j$ conflict, then $T_i$ is synchronized before $T_j$ iff $c_i < c_j$. To combine locking and timestamping, we must render their synchronization orders consistent.

Our algorithm uses MV timestamping to process read-only transactions (*queries*). The algorithm uses MV locking to process general transactions (*updaters*). Queries and updaters are assigned timestamps satisfying two properties:

(1) Let $T_i$ and $T_j$ be updaters. If $c_i < c_j$ then $\mathrm{TS}(i) < \mathrm{TS}(j)$.
(2) Let $T_q$ be a query and $T_i$ an updater. If $r_q[x_k] < w_i[x_i]$ then $\mathrm{TS}(q) < \mathrm{TS}(i)$.

A *consistent timestamp generator* is any means of assigning timestamps that satisfy these properties.

Our algorithm uses a *Lamport clock* to generate consistent timestamps. Recall the discussion of distributed systems from Section 2. A Lamport clock assigns a number to each event (called its *time)* subject to two conditions.

LC1. If $e$ and $f$ are events of the same process and $e$ happened before $f$, then $\mathrm{time}(e) < \mathrm{time}(f)$.
LC2. If $e$ is the event "process $P$ sends message $M$" and $f$ is the event "process $Q$ receives $M$," then $\mathrm{time}(e) < \mathrm{time}(f)$.

LC1 is easily achieved using clocks or counters local to each process. LC2 can be implemented by stamping each message with the local clock time when it was sent; if a process $Q$ receives a message whose time $t$ is greater than $Q$'s local time, $Q$ pushes its clock ahead to $t$.

LC1 and LC2 imply the following.

LC. Let $e$ and $f$ be events in a distributed system. If $e < f$ then $\mathrm{time}(e) < \mathrm{time}(f)$ [13].

LC is precisely the condition we need to generate consistent timestamps. When an updater $T_i$ is certified, the process that certifies it assigns $TS(i) = time(c_i)$. By LC, $c_i < c_j$ implies $time(c_i) < time(c_j)$; hence $TS(i) < TS(j)$ as desired. When a query $T_q$ begins executing, we make $TS(q)$ less than or equal to the current Lamport time. So for all reads $r_q[x_k]$, $TS(q) < time(r_q[x_k])$. Consider any write $w_i[x_i]$ such that $r_q[x_k] < w_i[x_i]$. By locking property L1 (see Section 5), $w_i[x_i] < c_i$, so by transitivity $r_q[x_k] < c_i$. By LC this implies $time(r_q[x_k]) < time(c_i)$; hence $TS(q) < TS(i)$ as desired.

We now describe the algorithm in detail.

(1) The system maintains a Lamport clock.

(2) Updaters use the MV locking algorithm of Section 5.

(3) When an updater $T_i$ is certified, the system assigns $TS(i) = time(c_i)$. This timestamp is transmitted to all versions that $T_i$ wrote. Thus, certified versions have timestamps, but uncertified versions do not.

(4) When a query $T_q$ begins executing, the system makes $TS(q)$ less than or equal to the current time.

(5) Consider any read by $T_q$, $r_q[x]$. As in Section 4, we want to translate this into $r_q[x_k]$ where $x_k$ is the version of $x$ with the largest timestamp less than $TS(q)$. But, some care is needed since uncertified versions do not have timestamps. Let $t$ be a lower bound on the possible timestamps of any uncertified $x$ versions. For instance, let $t = \min\{time(cl_i[x_i]) \mid x_i \text{ is uncertified}\}$. Since $cl_i[x_i] < c_i$, $time(cl_i[x_i])$ is a lower bound on $time(c_i) = TS(i)$; therefore $t$ is a lower bound on the timestamps of any uncertified $x_i$.

Consider $r_q[x]$ again. If $x$ has no uncertified versions, or if $TS(q) < t$, then $r_q[x]$ reads the version $x_k$ of $x$ with the largest timestamp less than $TS(q)$; else $r_q[x]$ *waits* until the condition is satisfied. (This will eventually happen.)

The log properties induced by the algorithm are a simple combination of the properties induced by MV timestamping and locking. The correctness proof is similar to those in Sections 4 and 5.

MULTIVERSION MIXED METHOD THEOREM. *All logs produced by the MV mixed method are* 1-*SR.*

Prime's algorithm differs from ours in two respects. Most importantly, Prime's algorithm does not use explicit timestamps. All certify events are $<$-related, that is, $c_1, \ldots, c_n$ are totally ordered. The algorithm maintains a list, CL, of all transactions that have been certified; when $T_i$ is certified, its identifier, $i$, is included in CL. When a query $T_q$ begins executing, it makes a copy of CL, denoted $CL(q)$. When $T_q$ issues a read, $r_q[x]$, it reads $x_k$ where $x_k$ is the latest version (with respect to $\ll$) of $x$ such that $k \in CL(q)$. We can analyze this behavior as a special case of our mixed method. Imagine that each updater $T_i$ is assigned a timestamp equal to its place in the certification total order, that is, $TS(i) = t$ iff $T_i$ is the $t$th transaction to be certified. Imagine that $T_q$ is assigned the timestamp $TS(q) = |CL(q)| + \varepsilon$, for $0 < \varepsilon < 1$. This is a consistent way of assigning timestamps. If we now run $T_q$ under our algorithm, it reads the same versions as under Prime's algorithm. Since our algorithm is 1-SR, so is Prime's.

The other difference is that Prime uses a restricted form of multiversion locking for updaters, namely two-phase locking [8]. Write operations set write-locks, so

that no data item ever has more than one uncertified version. And, once $T_i$ writes $x$, no updater $T_j$ reads $x$ until $T_i$ is certified, and vice versa. Consequently, every updater can be certified as soon as it finishes executing.

The net effect is that queries and updaters are totally decoupled. Queries never delay or cause the abort of updaters, and updaters never delay or cause the abort of queries.

Prime's algorithm is most naturally implemented in a centralized DBS because of the need to totally order certify events.

The following variant is more suitable for a distributed DBS.

(1) The system maintains a Lamport clock.
(2) Updaters use two-phase locking, hence they can be certified as soon as each finishes executing. The system assigns $\mathrm{TS}(i) = \mathrm{time}(c_i)$, as in the general algorithms.
(3) Queries are processed using timestamps, exactly as in the general algorithm.

This algorithm decouples queries and updaters almost as fully as Prime's algorithm. Queries never delay or abort updaters, and updaters never abort queries. But an updater can delay a query under one condition: If a query $T_q$ reads $x$, updater $T_i$ has a certify-lock on $x$, and $\mathrm{TS}(q)$ is greater than the time of that certify-lock, then $T_q$ must wait until $T_i$ certifies $x$.

## 7. CONCLUSION

This paper has studied the concurrency control problem for multiversion databases. Multiversion databases add a new aspect to concurrency control. Transactions issue operations that specify data items (e.g., read($x$), write($x$)); the system must *translate* these into operations that specify versions. In a single-version database, concurrency control correctness depends on the *order* in which reads and writes are processed. In a multiversion database, correctness depends on *translation* as well as order.

We have extended concurrency control theory to account for the translation aspect of multiversion databases. The main idea is *one-copy serializability*: an execution of transactions in a multiversion database is one-copy serializable (*1-SR*) if it is equivalent to a serial execution of the same transactions in a single-version database. A multiversion concurrency control algorithm is correct if all of its executions are 1-SR. We derived effective necessary and sufficient conditions for an execution to be 1-SR; these condition use the concept of *version order*. We gave a graph structure, *multiversion serialization graphs* (*MVSGs*), that helps check these conditions. Once a version order is fixed, an execution is 1-SR iff its MVSG is acyclic. MVSGs are analogous to the serialization graphs widely used in single-version concurrency control theory.

We applied the theory to three multiversion concurrency control algorithms. One algorithm uses timestamps, one uses locking, and one combines locking with timestamps. The timestamping algorithm is Reed's [17]. The locking algorithm was inspired by (and generalizes) the work of Bayer et al. [1, 2] and Stearns and Rosenkrantz [20]. The combination algorithm generalizes an algorithm developed by Prime Computer, Inc. [7] and used by Computer Corporation of America [6].

## REFERENCES

1. Bayer, R., Elhardt, E., Heller, H., and Reiser, A. Distributed concurrency control in database systems. In *Proc. 6th Int. Conf. Very Large Data Bases* (Montreal, Oct. 1–3, 1980),

ACM, New York, 1980, pp. 275–284.
2. BAYER, H., HELLER, H., AND REISER A. Parallelism and recovery in database systems. *ACM Trans. Database Syst. 5,* 2 (June 1980), 139–156.
3. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13,* 2 (June 1981), 185–221.
4. BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. SE-5,* 3 (May 1979), 203–215.
5. CASANOVA, M. A. *The Concurrency Control Problem of Database Systems.* Lecture Notes in Computer Science, vol. 116, Springer-Verlag, New York, 1981. (Originally published as Tech. Rep. TR-17-79, Center for Research in Computing Technology, Harvard University, 1979.)
6. CHAN, A., FOX, S., LIN, W. T. K., NORI, A., AND RIES, D. R. The implementation of an integrated concurrency control and recovery scheme. In *Proc. 1982 ACM SIGMOD Conf. Management of Data* (Orlando, Fla., June 2–4, 1982), M. Schkolnick, Ed., ACM, New York, 1982, pp. 184–191.
7. DUBOURDIEU, D. J. Implementation of distributed transactions. In *Proc. 1982 Berkeley Workshop on Distributed Data Management and Computer Networks,* pp. 81–94.
8. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19,* 11 (Nov. 1976), 624–633.
9. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, San Francisco, 1979.
10. GRAY, J. N. Notes on database operating systems. In *Operating Systems: An Advanced Course,* Lecture Notes in Computer Science, vol. 60, Springer-Verlag, New York, 1978, pp. 393–481.
11. HOLT, R. C. Some deadlock properties of computer systems. *ACM Comput. Surv. 4,* 3 (Sept. 1972), 179–196.
12. KING, P. F., AND COLLMEYER, A. J. Database sharing—an efficient mechanism for supporting concurrent processes. In *Proc. 1974 NCC,* AFIPS Press, Montvale, N.J., 1974.
13. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July 1978), 558–565.
14. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26,* 4 (Oct. 1979), 631–653.
15. PAPADIMITRIOU, C. H., AND KANELLAKIS, P. C. On concurrency control by multiple versions. In *Proc. ACM Symp. Principles of Database Systems* (Los Angeles, March 29–31, 1982), ACM, New York, 1982, pp. 76–82.
16. PAPADIMITRIOU, C. H., BERNSTEIN, P. A., AND ROTHNIE, J. B., JR. Some computational problems related to database concurrency control. In *Proc. Conf. Theoretical Computer Science,* (Waterloo, Ontario, Aug. 1977).
17. REED, D. Naming and synchronization in a decentralized computer system. Tech. Rep. MIT/LCS/TR-205, Dept. Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1978.
18. ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, P. M., II System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3,* 2 (June 1978), 178–198.
19. SILBERSCHATZ, A. A multi-version concurrency control scheme with no rollbacks. In *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing* (Ottawa, Canada, Aug. 18–20, 1982), ACM, New York, 1982, pp. 216–223.
20. STEARNS, R. E., AND ROSENKRANTZ, D. J. Distributed database concurrency controls using before-values. In *Proc. 1981 ACM SIGMOD Conf. Management of Data,* ACM, New York, 1981, pp. 74–83.
21. STEARNS, R. E., LEWIS, P. M., II, AND ROSENKRANTZ, D. J. Concurrency controls for database systems. In *Proc. 17th Symp. Foundations of Computer Science,* IEEE, New York, 1976, pp. 19–32.

Received July 1982; revised November 1982; accepted December 1982