JOURNAL OF COMPUTER AND SYSTEM SCIENCES 29, 207–224 (1984)

Multi-version Concurrency Control Scheme for a Database System*

SHOJIRO MURO

Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto 606, Japan

TIKO KAMEDA

Department of Computing Science, Simon Fraser University, Burnaby, British Columbia V5A 1S6, Canada

AND

TOSHIMI MINOURA

Department of Computer Science, Oregon State University, Corvallis, Oregon 97331

Received April 13, 1982; revised July 10, 1983

A concurrency control scheme using multiple versions of data objects is presented which allows increased concurrency. The scheme grants an appropriate version to each read request. Transactions issuing write requests which might destroy database integrity are aborted. It is precisely stated when old versions can be discarded and how to eliminate the effects of aborted transactions is described in detail. The scheduler outputs only (*multi-version*) *ww-serializable* histories which preserve database consistency. It is shown that any "*D*-serializable" history of Papadimitriou (*J. Assoc. Comput. Mach.* **26** (4) (1979), 631–653) (or "conflict-preserving serializable log" of Bernstein *et al.*, *IEEE Trans. Software Engrg.* **SE-5** (3) (1979), 203–216) is *ww*-serializable.

1. INTRODUCTION

In a typical database system, many users access shared data concurrently. Unless some kind of discipline is imposed on the activity of user transactions, data in the system may be modified in some unintended way. The reader is referred to [13] for problems that may arise. A *concurrency control* scheme should realize a high level of concurrency without destroying database consistency caused by undesirable

* This work was partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant Nos. A5240 and A4315.

0022-0000/84 $3.00

interactions among transactions. Many schemes for concurrency control have been proposed and analized [2, 3, 7, 9, 19, 22, 24, 26, 27].

It appears that the idea of using multiple *versions* of data objects was first tried in a Honeywell file system [14], and that the first theoretical analysis of a multiversion database system appeared in [26]. If we keep multiple versions of each data object, there is more likelihood of being able to grant read requests that arrive "too late," since older versions are saved for future read requests. Consider the following *history*:

$$W_1[X]\, R_2[X]\, W_2[Y]\, R_1[Y],$$

where $R_i[X]$ and $W_i[X]$ denote read and write operations, respectively, on object X by transaction T_i. Since T_2 reads the value of X from T_1 and T_1 reads the value of Y from T_2, the above history is not *equivalent* [9] to a *serial* history. The problem here is that $R_1[Y]$ arrives too late, i.e., after $W_2[Y]$. If we keep the initial value of Y as well as the new value written by $W_2[Y]$, then $R_1[Y]$ can access the old value, and the history now becomes equivalent to a serial history.

A family of multi-version concurrency control schemes are investigated and a number of important concepts and results are presented in [26]. Bayer, *et al.* [3] and Kessels [17] tried to make use of the fact that most database systems maintain two versions, (the old and the new versions) of each object for recovery reasons, while a transaction is modifying it. In the concurrency control scheme of [3], a read request by a transaction can be always granted. A read operation is given either the old or the new version, depending on the state of the object with respect to updating. This clearly increases concurrency to some degree.

In Section 2, we describe the model of a database system used in this paper. It is similar to the model used by Stearns, *et al.* [26], but we do *not* assume that a transaction must read an object in order to update it. Also, unlike their model we *do* allow versions created by unterminated transactions to be read by other transactions. This allows further concurrency at the price of increasing the probability of future abortion. As in [22], we allow each transaction to make at most one read and at most one write access to each object, but unlike the "two-step" model [22] we allow read (and similarly write) operations of a transaction to be performed at different times.

Section 3 introduces new concepts such as *ww-equivalence* and *ww-serializability*, which are stronger than the usual equivalence and *serializability* [22]. It is shown in [15] that *ww*-serializability in the single-version environment coinsides with "conflict-preserving serializability" [4, 5] and "*D*-serializability" [22]. *Ww*-equivalence and *ww*-serializability in this paper refer to multi-version histories. A useful tool called the *history graph*, which represents information regarding the "reads-from" relation and the "version-order" [6] in a history, are also defined here.

In Section 5, we present a multi-version concurrency control algorithm (Algorithm MV) which schedules operations of user transactions by maintaining and updating relevant subgraphs of the history graph and the *dependency graph*. One or more transactions are aborted if a write operation would create a cycle in the depen-

dency graph. We prove that a read operation can be always granted (i.e., there is always an appropriate version to be read without violating *ww*-serializability). The proof that Algorithm MV preserves database consistency is given in Section 6.

Main contributions in this paper consist of the following:

(1) Extension of the model in [26]. A transaction can update an object without first reading it. We also allow versions written by unterminated transactions to be read by other transactions.

(2) Introduction of the history graph. This graph faithfully records "essential" information on transaction execution, enabling backing up of transactions. The graph can be embedded in a database itself without much space overhead.

(3) Definition of multiversion *ww*-serializability. This is an extension of the notion of "conflict-preserving serializability" [4, 5] and "*D*-serializability" [22] to the multiversion case.

(4) A multiversion concurrency control sheme based on the history graph and the dependency graph. We clearly indicate when transactions can be *committed* and when old versions can be discarded. We also give a clear exposition of how to abort a transaction and how the effect of a transaction abortion propagates, necessitating the abortion of other transactions.

(5) Discussion of "checkpoint transactions" [11].

A practical multiversion database system based on timestamping has recently been built and it is claimed [8] experience with it indicates that keeping multiple versions can be practical in terms of time and space overhead.

This paper is a revision of our earlier report [20]. Some interesting papers on multiversion database systems have appeared since we submitted this paper for publication [6, 23]. We have added references to them in the Conclusion.

2. Database System Model and Histories

In this section, we shall describe the database system model used in this paper. A database system consists of a set **D** of *objects* (or *data items*), a set $\mathbf{T} = \{T_0, T_1, ..., T_n\}$ of *transactions*, and a *scheduler*. A transaction starts with a BEGIN request and ends with a TERMINATE request. Other steps of a transaction form a sequence of read and write operations. A *read operation* $R_i[X]$ of transaction T_i returns a value of object X, and a *write operation* $W_i[X]$ of transaction T_i creates a new value for X. Each object is accessed by at most one read operation and at most one write operation of each transaction. If a transaction T_i both reads and writes an object X, then $R_i[X]$ occurs before $W_i[X]$ in T_i, since T_i need not read what it has written. T_0 is the *initial transaction*, which is a fictitious write-only transaction that "writes" all objects in **D**. All (write) operations of T_0 take place before any operation of other transactions.

We say that transactions execute *concurrently* if the operations of the transactions are interleaved.[1] In order to avoid concurrent execution of transactions which destroys database consistency the execution of some operations may have to be delayed or rejected by the scheduler of the system. A *concurrency control algorithm* is the specification of a scheduler.

In a multiversion database system, each write operation on an object X creates a new *version* of X, instead of overwriting the current value of X as in a single-version database system. This gives a "late" read operation the chance to read a value which would have been erased in a single-version system. Thus, a multiversion history must specify which particular version each read operation reads. Let $OP(\mathbf{T})$ denote the set of all read and write operations of a set $\mathbf{T}$ of transactions. A multiversion history over $\mathbf{T}$ is defined as follows.

DEFINITION 2.1. A (multiversion) *history* over a set $\mathbf{T}$ of transactions is a triple $h = (OP(\mathbf{T}), <_h, F_h)$, where $<_h$ is a total order on $OP(\mathbf{T})$ and F_h is a mapping from the set of read operations $\{R_i[X] \mid R_i[X] \in OP(\mathbf{T})\}$ to the set of write operations $\{W_j[X] \mid W_j[X] \in OP(\mathbf{T})\}$ such that $F_h(R_i[X]) <_h R_i[X]$ for each $R_i[X]$. The total order $<_h$ orders all operations of the initial transaction T_0 before any other operation.

In this paper, a history means a multiversion history unless otherwise stated. Let $h = (OP(\mathbf{T}), <_h, F_h)$ be a history. For two operations A and B in $OP(\mathbf{T})$, we say that A *precedes* B in h if $A <_h B$. Intuitively, $<_h$ represents the time order in which the operations in $OP(\mathbf{T})$ are executed,[2] and for each read operation, F_h identifies the write operation which created the version that is read. The condition $F_h(R_i[X]) <_h R_i[X]$ requires that a read operation read an existing version; i.e., it cannot read a version to be created subsequently. In order to represent a history we shall use a visual representation given by the following definition.

DEFINITION 2.2. The *linear representation* of a history $h = (OP(\mathbf{T}), <_h, F_h)$ is one in which the operations of $OP(\mathbf{T})$ are arranged linearly from left to right in the order of $<_h$. Let $W_i[X], R_j[X] \in OP(\mathbf{T})$. An arrow is drawn from $W_i[X]$ to $R_j[X]$ iff $F_h(R_j[X]) = W_i[X]$.

Note that the arrows are all directed from left to right, reflecting $F_h(R_i[X]) <_h R_i[X]$.

EXAMPLE 2.1. The linear representation of a history $h = (OP(\mathbf{T}), <_h, F_h)$ is given below.

$$W_0[\mathbf{D}]\, R_1[X]\, R_2[Z]\, W_3[X]\, W_1[Z]\, R_3[Y]\, R_3[Z]\, W_4[X]\, R_2[X]\, W_3[Z]\, W_2[Y].$$

[1] We assume that the execution intervals of the operations accessing the same object do not overlap, but that other operations may overlap in time. Also, see the next footnote.

[2] If the execution intervals of two operations overlap, the operations are ordered according to their initiation time.

The leftmost $W_0[\mathbf{D}]$ stands for the write operations on all objects in $\mathbf{D}$ by the initial transaction T_0. The total order $<_h$ is represented by the order (left to right) in which the operations of $OP(\mathbf{T})$ appear. The arrow from $W_3[X]$ to $R_2[X]$, for instance, indicates that $F_h(R_2[X]) = W_3[X]$, i.e., T_2 reads the value of X written by T_3. We have omitted arrows enamating from $W_0[\mathbf{D}]$ in this example.

The following definition provides a link between this model and a single-version model.

DEFINITION 2.3. A history $h = (OP(\mathbf{T}), <_h, F_h)$ is said to be *normalized* if, for each $R_i[X] \in OP(\mathbf{T})$, there is no write operation $W_k[X]$ such that $F_h(R_i[X]) <_h W_k[X] <_h R_i[X]$.

Intuitively, in a normalized history, each read operation $R_i[X]$ reads the result of the "most recent" write operation on X, i.e., the nearest write operation to the left in the linear representation. A normalized history can be interpreted as a history in a single-version database system. In fact, if the arrows are removed from the linear representation of a normalized history, the conventional representation of a single-version history results [4, 5, 22]. Arrows are omitted because it is understood that each read operation reads the result of the "most recent" write operation.

DEFINITION 2.4. Two histories $h = (OP(\mathbf{T}), <_h, F_h)$ and $h' = (OP(\mathbf{T}), <_{h'}, F_{h'})$ over a set $\mathbf{T}$ of transactions are said to be *equivalent*, written $h \equiv h'$, if $F_h = F_{h'}$.

Thus, if h and h' are equivalent, then for each $X \in \mathbf{D}$ and any pair of transactions T_i, T_j, transaction T_i reads a version of X from T_j in h iff T_i reads a version of X from T_j in h'. Equivalence of two arbitrary histories can be easily tested using the above definition. In the next section we consider equivalence with some constraints.

Serializability of a history is widely accepted as a useful criterion for a "correct" history [4, 5, 9, 22]. In the rest of this section we extend some serializability-related concepts to the multiversion environment.

DEFINITION 2.5. A normalized history $h = (OP(\mathbf{T}), <_h, F_h)$ is called *serial* if there is a total order $\ll$ on the set $\mathbf{T}$ such that for any two distinct transactions T_i and T_j in $\mathbf{T}$, if $A <_h B$ then $T_i \ll T_j$, where A (B) is any operation of T_i (T_j).

In the linear representation of a serial history, all operations of each transaction appear consecutively, and there is an arrow to each read operation from the nearest write operation to the left that accesses the same object. A serial history is a paradigm of a "correct" history. It is assumed that a single transaction executed alone preserves database consistency. Therefore a sequence of such executions, i.e., a serial history, also preserves database consistency. This observation motivates the following definition.

DEFINITION 2.6. A multiversion history h is said to be *serializable* if there exists a serial history h' that is equivalent to h.

Papadimitriou has shown that the test for serializability is *NP*-complete [12], even for his "two-step" transaction model in the single-version case [21, 22]. Since the single-version case is a simple "restriction" [12] of the multiversion case, it follows that the test for serializability in the multiversion case is also *NP*-hard.

3. *ww*-Serializability and History Graph

In this section we first introduce the concepts of *ww-equivalence* and the *ww-serializable* history. We then introduce the *history graph* and discuss its properties.

Definition 3.1. Let $h = (OP(\mathbf{T}), <_h, F_h)$ and $h' = (OP(\mathbf{T}), <_{h'}, F_{h'})$ be two histories over a set $\mathbf{T}$ of transactions. The histories h and h' are said to be *equivalent with the ww-constraints* (*ww-equivalent*, for short, written $h \equiv [ww]\, h'$), if $F_h = F_{h'}$ and, for each object X and each pair of write operations on X, $W_i[X] <_h W_j[X]$ holds whenever $W_i[X] <_{h'} W_j[X]$.

Given the linear representations of h and h', *ww*-equivalence between them can be tested efficiently as follows. For each $X \in \mathbf{D}$, check if the write operations on X appear in the same order in h and h'. This can be done in time proportional to $\|h\|$, i.e., the length of (or the number of operations in) h. Then check if $F_h = F_{h'}$. It is easy to see that the total time required is linear in $\|h\|$.

Lemma 3.1. *Let $h = (OP(\mathbf{T}), <_h, F_h)$ be a history over a set $\mathbf{T}$ of transactions. There exists a normalized history $h' = (OP(\mathbf{T}), <_{h'}, F_{h'})$ over $\mathbf{T}$ such that $h' \equiv [ww]\, h$.*

Proof. We shall construct h' starting with $h' = h$, i.e., $<_{h'} = <_h$ and $F_{h'} = F_h$. If there is a read operation $R_i[X]$ in h' such that $F_{h'}(R_i[X]) <_{h'} W_j[X] <_{h'} R_i[X]$ for some write operation $W_j[X]$, then move $R_i[X]$ left in the linear representation of h' to just before $W_j[X]$, i.e., modify $<_{h'}$ so that $R_i[X] <_{h'} W_j[X]$. Repeat this process until no such $R_i[X]$ exists for any i or X. The resulting sequence represents the desired normalized history h', since we clearly have $F_{h'} = F_h$. ∎

For each object X, its initial value (written by T_0) is defined to be *version* 0, and a new version of X is created by each succeeding write operation on X. The *version number* of a version of X created by a write operation $W_i[X]$ in a history h is denoted by $\#h(W_i[X])$. After the initial version, the subsequent write operations on X generate versions with successively higher integers as their version numbers. Intuitively, $\#h(W_i[X])$ is the order of appearance of $W_i[X]$ among the write operations on X in the linear representation of h. It follows from the definition of *ww*-equivalence that if $h \equiv [ww]\, h'$ then $\#h(W_i[X]) = \#h'(W_i[X])$ for each write operation $W_i[X] \in OP(\mathbf{T})$. Therefore, in discussing two *ww*-equivalent histories, we may assume that the version numbers are fixed.

DEFINITION 3.2. A history h is said to be *serializable with ww-constraints* (*ww-serializable*, for short) if there exists a serial history h' such that $h \equiv [ww]\, h'$.

As we show in the next section, *ww*-serializability of a history can be tested efficiently.

Remark. As mentioned in Section 2, a normalized history can be interpreted as a history for a single-version database system. In this connection, it turns out that every "D-serializable" history [22] or "conflict-preserving serializable log" [4, 5] is a normalized *ww*-serializable history. We assume here that a read (write) operation of the model in [4, 5, 22] is changed into a sequence of consecutive read (write) operations, one for each object in the "read (write) set." If no write operation is allowed to precede any read operation in each transaction, then every normalized *ww*-serializable history can be interpreted as a "D-serializable" (or "conflict-preserving") history [15].

We now introduce a graphical representation for a history. This graph will play a key role in subsequent discussions. Let $h = (OP(\mathbf{T}), <_h, F_h)$ be a history over a set $\mathbf{T}$ of transactions. In order to represent h, we construct a labelled bipartite graph, $\mathrm{HG}(h) = (N, A)$, called the *history graph*, where N and A are the set of nodes and set of arcs, respectively. Intuitively, for each transaction T_i in $\mathbf{T}$, $\mathrm{HG}(h)$ represents the versions accessed (i.e., read or written) by T_i. N consists of the following nodes:

(N1) For each transaction T_i in $\mathbf{T}$, there is a node ("transaction node") with label T_i.

(N2) For each pair $[X, v]$ such that $X \in \mathbf{D}$ and v is the version number for X, there is a node ("version node") with label $[X, v]$.

The arcs in A are given as follows:

(A1) $([X, v], T_i) \in A$ iff $\#h(F_h(R_i[X])) = v$.

(A2) $(T_i, [X, v]) \in A$ iff $\#h(W_i[X]) = v$.

Suppose $h = (OP(\mathbf{T}), <_h, F_h)$ is *ww*-equivalent to $h' = (OP(\mathbf{T}), <_{h'}, F_{h'})$. Then T_i writes the version v of an object X in h, iff T_i writes the version v of X in h'. Similarly, T_i reads the version v of X in h, iff T_i reads the version v of X in h'. Thus we have the following lemma.

LEMMA 3.2. *Let h and h' be two histories over a set $\mathbf{T}$ of transactions. $h \equiv [ww]\, h'$ is and only if $\mathrm{HG}(h) = \mathrm{HG}(h')$.*

4. DEPENDENCY GRAPH

We now develop a method for testing *ww*-serializability of a history. For a set of transactions $\mathbf{T} = \{T_0, T_1, \ldots, T_n\}$, let $h = (OP(\mathbf{T}), <_h, F_h)$ be a given history. We construct a directed graph associated with h.

DEFINITION 4.1. The *dependency graph* $DG(h) = (\mathbf{T}, B)$, associated with a history $h = (OP(\mathbf{T}), <_h, F_h)$ over a set $\mathbf{T}$ of transactions is a directed graph with node set $\mathbf{T}$. The set of arcs of $DG(h)$, B, is defined as follows:

$$(T_i, T_j) \in B \qquad \text{for } i \text{ and } j \text{ with } i \neq j,$$

iff any of the following conditions holds for some $X \in D(\mathbf{T})$:

(1) (*ww*-constraint). There exist two operations $W_i[X]$ and $W_j[X]$ in $OP(\mathbf{T})$ such that $\#h(W_j[X]) = \#h(W_i[X]) + 1$.

(2) (Exclusion [15]). There exist two operations $R_i[X]$ and $W_j[X]$ in $OP(\mathbf{T})$ such that $\#h(W_j[X]) = \#h(F_h(R_i[X])) + 1$.

(3) (Reads-from). There exist two operations $W_i[X]$ and $R_j[X]$ in $OP(\mathbf{T})$ such that $F_h(R_j[X]) = W_i[X]$.

We call arc $(T_i, T_j) \in B$ a *ww-arc*, an *exclusion arc*, and a *reads-from arc*, respectively, if conditions (1), (2), and (3) above hold. Exclusion arcs and *ww*-arcs are sometimes called *constraint arcs*. Now let h be a serial history and let $T_i \ll T_j$ for $T_i, T_j \in \mathbf{T}$ (see Definition 2.5). According to the above definition, (T_j, T_i) cannot be an arc of $DG(h)$. Therefore $DG(h)$ must be acyclic for a serial history h.

LEMMA 4.1. *Let* $\mathbf{T} = \{T_0, T_1, ..., T_n\}$, *and let* $h = (OP(\mathbf{T}), <_h, F_h)$ *and* $h' = (OP(\mathbf{T}), <_{h'}, F_{h'})$ *be two histories over* $\mathbf{T}$. *If* $HG(h) = HG(h')$, *then* $DG(h) = DG(h')$.

Proof. First note that $DG(h)$ and $DG(h')$ have the same set of nodes. If $HG(h) = HG(h')$, then we have $\#h = \#h'$ and $F_h = F_{h'}$ by Lemma 3.2. Therefore the arcs due to the conditions (1)–(3) in Definition 4.1 are identical for $DG(h)$ and $DG(h')$. ∎

We now prove an important theorem.

THEOREM 4.1. *A history h is ww-serializable if and only if* $DG(h)$ *is acyclic.*

Proof. First assume that h is *ww*-serializable. Then, by definition, there is a serial history h' such that $h \equiv [ww]\, h'$. By Lemma 3.2, we have $HG(h) = HG(h')$ and thus $DG(h) = DG(h')$ by Lemma 4.1. Since h' is a serial history, $DG(h')$ is acyclic.

In order to prove the "if part" of the theorem, assume that $DG(h)$ is acyclic. Let $T_0 \ll T_{p(1)} \ll T_{p(2)} \ll \cdots \ll T_{p(n)}$ be a total order obtained by *topologically sorting* [18] the set $\{T_0, T_1, ..., T_n\}$ of nodes of $DG(h)$, where $p(\cdot)$ is a permutation on the set $\{1, 2, ..., n\}$. Note that if $i < j$ then there is no path from $\mathbf{T}_{p(j)}$ to $\mathbf{T}_{p(i)}$ in $DG(h)$. For each i $(1 \leqslant i \leqslant n)$, let $\mathrm{Op}(i)$ be the sequence of operations of T_i. Concatenate these sequences to construct the linear representation of a serial history h': $\mathrm{Op}(0)\,\mathrm{Op}(p(1))\,\mathrm{Op}(p(2)) \cdots \mathrm{Op}(p(n))$. We claim that $h \equiv h'$, and therefore h is *ww*-serializable.

To prove $h \equiv h'$, let $W_i[X]$, $W_j[X] \in OP(\mathbf{T})$. We first show that $W_i[X] <_h W_j[X]$

iff $W_i[X] <_{h'} W_j[X]$. If $W_i[X] <_h W_j[X]$ then $T_i \ll T_j$ (by the *ww*-constraint), and $W_j[X]$ cannot appear before $W_i[X]$ in h'. Similarly, if $W_j[X] <_h W_i[X]$ then $W_i[X]$ must follow $W_j[X]$ in h'. It follows that for each v, version v of X is created by T_i in h iff it is created by T_i in h'. Let $\{T'_1, T'_2, ..., T'_k\} \subseteq \mathbf{T}$ be the set of all transactions that write X such that, for $v = 0, 1, 2, ..., k$, transaction T'_i writes version v of X both in h and h'. To complete the proof, suppose $T_j (\neq T'_{v+1})$ reads version v of X in h, i.e., $F_h(R_j[X]) = W'_v[X]$, where $W'_v[X]$ is a write operation of T'_v. Then there are arcs from T'_v to T_j (reads-from arc) and T_j to T'_{v+1} (exclusion arc) in DG(h), and therefore we have $T'_v \ll T_j \ll T'_{v+1}$. It follows that T_j reads version v of X in h', i.e., $F_{h'}(R_j[X]) = W'_v[X]$. The case $T_j = T'_{v+1}$ is easy to prove. We thus have $F_h = F_{h'}$, i.e., $h \equiv h'$. ∎

COROLLARY 4.1. *Let $D(\mathbf{T})$ be the set of objects that are accessed by any $T_i \in \mathbf{T}$. For a given history $h = (OP(\mathbf{T}), <_h, F_h)$, ww-serializability of h can be tested in $O(|D(\mathbf{T})| \cdot |\mathbf{T}|)$ time.*

Proof. For each object $X \in D(\mathbf{T})$, there are at most $2|\mathbf{T}|$ read and write operations which access X, according to the definition of a transaction. Construct from the linear representation of h the sequence of operations accessing only object X in $D(\mathbf{T})$. Such a sequence has at most $2|\mathbf{T}|$ operations. We now construct DG(h) as follows. First introduce the set $\mathbf{T}$ of nodes. For the sequence obtained for each object X, introduce an arc (T_i, T_j) if any of the three conditions in Definition 4.1 is satisfied.

The sequence associated with each object can be processed in $O(|\mathbf{T}|)$ time, which implies that at most $O(|\mathbf{T}|)$ arcs are introduced per object. Therefore the total number of arcs introduced is bounded by $O(|D(\mathbf{T})| \cdot |\mathbf{T}|)$. A cycle in DG($h$) can be tested in time linear in the number of nodes and arcs [1]. ∎

5. THE ALGORITHM

In this section, we discuss a multiversion concurrency control algorithm, called *Algorithm MV*, for the scheduler of a centralized database system. The input to the scheduler is a sequence of arriving requests from user transactions, including their BEGIN and TERMINATE requests. The BEGIN and TERMINATE requests from transaction T_i are denoted by $b(T_i)$ and $t(T_i)$, respectively.

5.1. *General Description*

In response to each input request, Algorithm MV tentatively updates the history graph and the dependency graph. Let $HG^* = (N, A)$ and $DG^* = (N', A')$, respectively, denote the subgraphs of the history graph and the dependency graph, which are maintained by the scheduler. They are initialized as follows, reflecting the situation where the initial transaction T_0 has just terminated:

$$N \leftarrow \{[X, 0] \mid X \in \mathbf{D}\} \qquad A \leftarrow \varnothing,$$

$$N' \leftarrow \varnothing; \qquad A' \leftarrow \varnothing.$$

Updating HG* is straightforward and is carried out as follows depending on the input request:

(1) For BEGIN request, $b(T_i)$: Create a transaction node T_i.

(2) For $W_i[X]$: Create a version node $[X, v]$, where $v =$ (currently largest version number of object X) + 1, and add an arc $(T_i, [X, v])$ to A.

(3) For $R_i[X]$: Create an arc $([X, v], T_i) \in A$, where v is the version number selected by the method to be described later in this section (see Theorem 5.1).

(4) For TERMINATE request, $t(T_i)$: If T_i satisfies the "deletion condition" (see Subsection 5.3), delete node T_i and possibly some version nodes (to be specified later) together with the arcs incident on them.

As we will see later, a write request is not always granted. If T_i is aborted as a result of the rejection of a write operation $W_i[X]$, the tentative changes made in response to $W_i[X]$ must be undone. Furthermore, other transactions may have to be backed up as a result, as we discuss in more detail below.

Lemma 4.1 implies that all information required to construct DG(h) is contained in HG(h). Therefore the updating of HG* described above can be translated into the updating of DG*. The details of the implementation of HG* will be discussed in Subsection 5.5.

There are still three points left unclear at this points:

(a) When should a write request be rejected?

(b) When can a transaction be "committed"?

(c) Which version should be given to a read request $R_i[X]$?

To "commit" a transaction means to give it a guarantee that it will never be aborted and its effects on the database will persist. We shall now address these questions one by one.

5.2. *Rejection of Write Requests and Abortion of Transactions*

When a write request $W_i[X]$ is received, both HG* and DG* are tentatively updated as described above. If a cycle is created in DG* as a result, then the partial history generated so far is not *ww*-serializable. That is to say, if all currently unterminated transactions immediately send TERMINATE requests, the generated history is not *ww*-serializable. Moreover, no matter what requests arrive in the future, the resultant history will not be *ww*-serializable. We reject $W_i[X]$, abort T_i, and undo the changes made to HG* and DG* in response to the requests made by T_i.

Remark 1. Note that as a result of aborting a transaction, some version nodes may be removed from HG*, and the version numbers of some objects may become nonconsecutive. If this situation arises, we must (at least in theory) renumber the affected versions and introduce new *ww*-arcs and exclusion arcs. However, the results of previous sections are still applicable without renumbering since, as the reader

recalls, version numbers were introduced to represent a total order among the versions of each object. Therefore the integers used need not be consecutive. (The conditions of Definition 4.1 must be restated accordingly.)

Furthermore, all transactions corresponding to the nodes of DG* reachable from T_i by reads-from arcs must be aborted, since they have read the versions to be discarded. This phenomenon is called the *domino effect* [25] or *cascading* [3]. In our scheme, this is the price we pay for increased concurrency.

Remark 2. Note that the procedure described above for eliminating a cycle in DG* is just one of many possible ways (perhaps the simplest). For example, a cycle in DG* can be broken by aborting transaction(s) other than T_i which issued the "offending" write request $W_i[X]$. Consider the tree, rooted at node T_i, whose nodes consist of those reachable via reads-from arcs from T_i in DG*. If T_i is aborted, then all other nodes of the tree must also be aborted. However, if we abort transactions one by one, starting at the leaf level of this tree, the cycle may disappear before we reach the root T_i. The transactions thus aborted will form a subset (possibly the whole set) of those which would be aborted by the above simple method.

Making $W_i[X]$ wait is a possible option only if other transactions are aborted [26]. Otherwise, a cycle will be created by $W_i[X]$ no matter how long $W_i[X]$ waits. This is the reason why we abort T_i. However, if the aborted transaction is immediately restarted, the so-called *cyclic restart* [26] may occur. For a discussion of cyclic restarts, including methods to cope with them, the reader is referred to [10, 26].

Note also that a complete abortion of a transaction may not be necessary, but a partial backing up may suffice. However, we use abortion for ease of exposition. All information needed for a partial backup is readily available in HG*. In general, we should salvage as many operations that have been performed as possible. At the same time, the overhead for determining which operations to undo for this purpose should also be taken into account.

5.3. *Commit and Deletion Conditions*

Let RF* be a partial graph of DG*, obtained from DG* by deleting all constraint arcs from it, leaving only the read-from arcs. Since we keep DG* acyclic, RF* is a fortiori acyclic. Let T_j be a *predecessor* of T_i in RF* (i.e., there is a path from node T_j to node T_i). Then transaction T_i has received some information from transaction T_j either directly, or indirectly via other transactions.

DEFINITION 5.1. Transaction T_i is said to satisfy the *commit condition* if it has sent its TERMINATE request $t(T_i)$ and all the predecessor transactions, if any, of node T_i in RF* have been committed.

When a transaction satisfies the commit condition. It means that this transaction will never be aborted in the future as a result of the domino effect, since all transactions from which it has received some information have already been committed. If we keep track of all information about the history forever, the total storage space

needed by the scheduler will grow without bounds. Therefore we delete nodes and arcs from HG* and DG* that are no longer needed. We call the transactions that are in the current HG* and DG* *open* transactions [26]. A node of a directed graph is called a *source* if it has no incoming arc.

DEFINITION 5.2. Transaction T_i is said to satisfy the *deletion condition* if it has sent its TERMINATE request $t(T_i)$ and node T_i is a source in DG*.

A transaction satisfying the commit condition may in addition satisfy the deletion condition. If T_i satisfies the deletion condition, we delete some nodes and arcs from HG* and DG* by the rules given below. (See Lemma 6.1 for justification.)

(D1) Delete node T_i from HG* together with the arcs incident on it (both incoming and outgoing arcs).

(D2) For each $X \in \mathbf{D}$ such that T_i has written a version of the form $[X, v]$, delete the version node $[X, v']$ in HG* with $v' < v$.

(D3) Delete node T_i from DG* together with its outgoing arcs.

Let $T_1', T_2', \ldots, T_k'$ be all the open transactions that have written versions of X, i.e., $[X, v_1], [X, v_2], \ldots, [X, v_k]$, where $v_1 < v_2 < \cdots < v_k$. Because of *ww*-constraints, there is an arc (T_i', T_{i+1}') in DG* for $i = 1, 2, \ldots, k-1$. Therefore they can satisfy the deletion condition only in the order, $T_1', T_2', \ldots, T_k'$. If there is an open transaction T_0' which read a previous version $[X, v_0]$ with $v_0 < v_1$, then there is an arc (T_0', T_1') in DG* due to the exclusion constraint (Sect. 4). T_1' can satisfy the deletion condition only after T_0' has been deleted. Rule (D2) makes sure that when T_i' $(i \geqslant 1)$ satisfies the deletion condition, the version $[X, v_{i-1}]$ is deleted. It can be shown that each time (D2) is applied, exactly one version is deleted. It is seen that versions are deleted in the order they were created (unless they are deleted by transaction abortion), and that exactly one version node (per object) with no incoming arc is always kept in the current HG*. The following lemma follows easily from this observation.

LEMMA 5.1. *For each object $X \in \mathbf{D}$, the only version node of the form $[X, v]$ with no incoming arc in* HG* *is the version with the largest version number among those written by the deleted transactions. (We consider T_0 as a deleted transaction.)*

The above lemma is important in proving that a read request can be always granted (see the next subsection).

There are two possible ways in which a transaction newly satisfies the deletion condition:

(a) The scheduler receives a TERMINATE request $t(T_i)$.

(b) A predecessor transaction of T_i in DG* has been deleted either by rule (D3) or by abortion, making T_i a source in DG*.

In Algorithm MV, we delete the transactions satisfying the deletion condition whenever they are found.

5.4. *Processing Read Requests*

If a read operation $R_i[X]$ is received by the scheduler and accesses a version v of X, HG* and DG* must be updated. Arc $([X, v], T_i)$ is added to HG* and a reads-from arc and constraint arcs are added to DG*. Unlike in a single-version database system, the scheduler has a choice as to which version of X to allow $R_i[X]$ to access. The following theorem states that we can always grant a read request.

THEOREM 5.1. *Algorithm* MV *always grants a read request.*

Proof. Consider the current dependency graph DG* and a new read request $R_i[X]$. We shall show that there always exists a version of X such that no cycle is created in DG* if the version is read by $R_i[X]$.

Let $\ll$ be the total order among the transactions as a result of topologically sorting DG*. Let T_j be a transaction which has written a version of X such that $T_j \ll T_i$ and there is no other such transaction between T_j and T_i in the $\ll$ order. If the version created by $W_j[X]$ is given to $R_i[X]$, clearly no cycle results in DG*. If such a T_j does not exist in DG*, we give $R_i[X]$ the oldest version of X that is still maintained. No arc is created in DG* as a result, and hence no cycle results. ∎

The above proof finds just one appropriate version to be given to a read operation. The set of *all* versions, any one of which can be given to a read request $R_i[X]$, can be determined efficiently as reported in [10].

5.5. *Implementation*

Here we discuss only an implementation of HG* which facilitates its updating. Implementing DG* is straightforward. For each object $X \in \mathbf{D}$, we maintain a list of lists, all doubly linked, named VLIST(X) (see Fig. 1). Each list in VLIST(X) represents a version of X and is headed by a record with four fields: *version-number*, *value*, *written-by*, and *read-by*. The first two fields are self-explanatory. The third field contains the name of the transaction which wrote this version. The last field is a

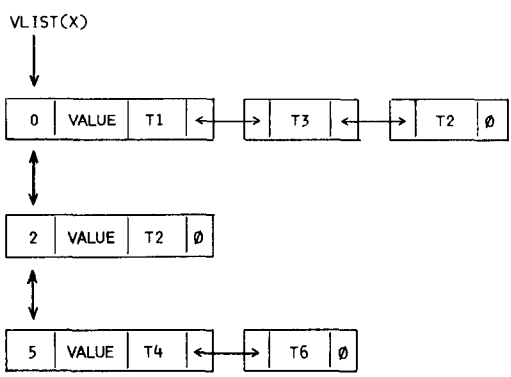

FIG. 1. Data structure for implementing HG*.

pointer to the linked list of all transactions that have read this version. Note that reads-from relation exists between the transaction in the third field and those in the linked list. Similarly, *ww*-constraint exists between two transactions which wrote adjacent versions. And finally, exclusion constraint exists between the transactions which read a version and the one which wrote the next newer version (see Fig. 1).

Although VLIST(X) for all $X \in \mathbf{D}$ provide all information about HG*, it is convenient to maintain another set of lists, two for each open transaction. The first list, WROTE(T_i), contains the pointers to all versions that T_i has created, and the second list, READ(T_i), contains the pointers to all versions that T_i has read. Actually a pointer in READ(T_i) does not point to a version itself, but rather to the element T_i in the list of transactions which read the version.

We illustrate how the deletion of a transaction T_i can be efficiently carried out using these data structures. First, for each element (pointer) in READ(T_i), delete the transaction pointed to by it. Next, for each element (pointer) in WROTE(T_i), delete the version pointed to by it. The abortion of one transaction may necessitate abortion of other transactions due to the domino effect. Each of the transactions reachable via reads-from arcs in DG* from an aborted transaction should also be deleted as above. Then, the reads-from relation, *ww*-constraint, and exclusion constraint in DG* must be reexamined.

5.6. *Special Cases*

There are two special cases of interest. First we consider read-only transactions. It follows from Theorem 5.1 that a read-only transaction is never aborted by Algorithm MV, unless one or more of its predecessors in RF* are aborted. Therefore, a read-only transaction which reads the oldest versions in HG* is never aborted, since it is a source and has no predecessor in RF*. Such a transaction can be used as a "check-point transaction" [11] which reads a consistent state of the entire database. Moreover, such a transaction does not cause abortion of other transactions, because it cannot be part of a cycle in DG* (it will remain a source in DG* as long as it is open).

The second special case is not really a restriction of our scheme, but rather a modification. It is obtained by allowing only one version for each object. We also add an additional condition on our transaction model that the set of all read operations and the set of all write operations of a transaction be each atomic and that the write operations follow the read operations in each transaction. Then we have the (single-version) "two-step" model used in [4, 7, 22]. For this model, we can show that each read (=atomic set of read operations) can be always granted and the implementation described in Subsection 5.5 compares favorably with that given in [7].

6. Correctness of the Algorithm

In this section we shall prove that Algorithm MV generates only *ww*-serializable histories. Our approach is to make use of Theorem 4.1 for an arbitrary history h

generated by Algorithm MV and to show that the dependency graph DG(h) is acyclic. Note that our algorithm deletes transaction nodes from DG* when they satisfy the deletion condition, so that it is not obvious whether the dependency graph would be acyclic if they were not deleted.

Let $\{T_1, T_2, ..., T_n\}$ be the set of all transactions received by the scheduler from the time the database system was initialized, excluding those aborted by the scheduler and removed from the system (unless they have reentered the system). Without loss of generality let $T_1, T_2, ..., T_m$, where $m \leqslant n$, be the transactions which satisfied the deletion condition up to the present time. Thus in the current DG* we have only the nodes corresponding to the open transactions, $T_{m+1}, T_{m+2}, ..., T_n$. These transactions are still in the system, either because they have not terminated, or because they are not sources in DG*. The following lemma shows that if the dependency graph is constructed for all of $T_1, ..., T_n$, it will be acyclic.

LEMMA 6.1. *Let* $\{T_1, T_2, ..., T_n\}$ *be the set of transactions which have been received by the scheduler so far, excluding those aborted. If they are scheduled using Algorithm* MV, *then the dependency graph for these transactions is acyclic.*

Proof. Assume that transactions $T_1, ..., T_m$ satisfied the deletion condition in that order and have been deleted from DG* maintained by the scheduler. Starting with the current DG*, which is acyclic, we shall first restore T_m back to it. We want to show that the resulting graph is also acyclic. Since T_m satisfied the deletion condition at the time it was deleted, it was a source then. By restoring a source to an acyclic graph we obtain another acyclic graph. Some new arcs which did not exist at the time of deletion may have to be introduced, since new transactions may have been received after the deletion. These arcs, if any, must be directed *from* node T_m to other nodes. Hence the dependency graph is acyclic. The lemma can be proved by induction on the number of transaction nodes restored as above. ∎

It follows from the proof of the above lemma that the dependency graph for $\{T_1, T_2, ..., T_m\}$ is acyclic. In other words, no matter what happens in the future, the partial history consisting of the operations of transactions $T_1, ..., T_m$ will not make the dependency graph cyclic. This is because the dependency subgraph defined by $T_1, ..., T_m$ is acyclic and future actions of the scheduler will never create an arc directed *to* any of $T_1, ..., T_m$.

It is now easy to prove the main theorem.

THEOREM 6.1. *Any history allowed by Algoritm* MV *is ww-serializable.*

Proof. Let $T_1, T_2, ..., T_n$ and the index m be as defined just before Lemma 6.1. Consider the case where $m = n$, i.e., all transactions are completed. It follows from Theorem 4.1 and Lemma 6.1 that the resultant history is *ww*-serializable. ∎

CONCLUSION

We have proposed a concurrency control scheme which maintains multiple versions of each data object. Keeping older versions enables a read request to be always granted, since an appropriate version to be read is always available. If it becomes apparent that a write operation must be rejected in order to preserve *ww*-serializability, we first abort the transaction that issued the write request. When a transaction is aborted, other transactions that have read the versions written by the aborted transaction must also be aborted. By referring to the partial history graph HG*, it is possible to salvage some computation of a transaction to be aborted, up to the first read operation that read an invalidated version.

The constraint arcs of the dependency graph were introduced more for expediency than for necessity. For example, if $W_j[X]$ arrives after $W_i[X]$, we let $W_j[X]$ create a newer version (i.e., version with a larger version number) than $W_i[X]$. However, it might be possible to let $W_j[X]$ create an "older" version than $W_i[X]$ in order to avoid a cycle that would otherwise be created. This possibility exists only if T_j did not read X, as reported in [10]. Reed [24] determines the version order by the time stamp of the *transaction* that issues the write request.

As stated in the Introduction, Stearns *et al.* [26] allow only versions written by terminated transactions to be read. We have removed this restriction, allowing any version to be read, provided this reading action does not cause a cycle in DG*. This feature increases the possibility of the domino effect. More quantitative analysis is required to see if or when the removal of this restriction is justified.

A multiversion *log* introduced in [6] is similar to a multiversion history, in the sense that the reads-from function F_h is a part of the definition. Papadimitriou *et al.* [23] use a different approach. The reads-from function is not a part of a *schedule.* In terms of our terminology, a schedule is simply a pair $(OP(\mathbf{T}), <_h)$, and the function F_h is called an *interpretation.* They introduce a set of schedules called DMVSR and investigate its properties. It can be shown [16] that if $h = (OP(\mathbf{T}), <_h, F_h)$ is *ww*-serializable then the schedule $(OP(\mathbf{T}), <_h)$ belongs to DMVSR, and that all schedules in DMVSR can be obtained from such histories. However, Algorithm MV may, given a schedule in DMVSR as an input, rearrange its operations before outputting them. The reason is that Algorithm MV chooses an arbitrary version to be given to a read request without knowing what other operations are to arrive in the future. An unfortunate choice here may necessitate abortion of some transactions in the future.

ACKNOWLEDGMENTS

Thanks are due to Professors I. F. Blake and E. G. Manning of the University of Waterloo without whose support this work would not have been possible. One of the authors, S. Muro, wishes to express his sincere appreciation to Professors T. Hasegawa of Kyoto University and T. Ibaraki of Toyohashi University of Technology for their support and encouragement. He also acknowledges stimulating discussions with Hide Tokuda and Gita Krishnan of the Computer Communications Networks Group at the University of Waterloo. Abdul Farrag of Simon Fraser University has also provided us with useful comments.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
2. M. M. ASTRAHAN, *et al.*, System R: Relational approach to database management, *ACM Trans. Database Systems* **1** (2) (1976), 97–137.
3. R. BAYER, H. HELLER, AND A. REISER, Parallelism and recovery in database systems, *ACM Trans. Database Systems* **5** (2) (1980), 139–156.
4. P. A. BERNSTEIN, D. W. SHIPMAN, AND S. W. WONG, Formal aspects of serializability in database concurrency control, *IEEE Trans. Software Engrg.* **SE-5** (3) (1979), 203–216.
5. P. A. BERNSTEIN AND N. GOODMAN, Concurrency control in distributed database systems, *ACM Comput. Surveys* **13** (2) (1981), 185–221.
6. P. A. BERNSTEIN AND N. GOODMAN, Concurrency control algorithms for multiversion database systems, *in* "Proceedings, ACM SIGACT/SIGOPS Sympos. Principles of Distributed Computing, August 1982," 209–215.
7. M. A. CASANOVA, The concurrency control problem for database systems, Lecture Notes in Computer Science No. 116, Springer-Verlag, Berlin, 1981.
8. D. J. DUBOURDIEU, Implementation of distributed transactions, *in* "Proceedings, 6th Berkeley Workshop on Dist. Data Manag. and Comput. Networks, February 1982," 81–94.
9. K. P. ESWARAN, J. N. GRAY, R. A. LORIE, AND I. L. TRAIGER, The notions of consistency and predicate locks in a database system, *Comm. ACM* **19** (11) (1976), 624–633.
10. A. A. FARRAG AND T. KAMEDA, "On Concurrency Control Using Multiple Versions," Technical Report TR **82-13**, Dept. Comput. Sci., Simon Fraser Univ., Burnaby, B.C., Canada, 1982.
11. M. J. FISCHER, N. D. GRIFFETH, AND N. A. LYNCH, Global states of a distributed system, *IEEE Trans. Software Engrg.* **SE-8** (3) (1982), 198–202.
12. M. R. GAREY AND D. S. JOHNSON, "Computers and Intractability—A Guide to the Theory of NP-Completeness," Freeman, San Francisco, Calif., 1979.
13. J. N. GRAY, Notes on data base operating systems, Lecture Notes in Computer Science No. 60, pp. 393–481, Springer-Verlag, Berlin, 1978.
14. Honeywell file management supervisor, Order No. DB54, Honeywell Information Systems, Inc., 1973.
15. T. IBARAKI, T. KAMEDA, AND T. MINOURA, SNOTS and its applications: Serializability theory made simple, Tech. Rept. No. 82-12, Dept. Comput. Sci., Simon Fraser Univ., Burnaby, B.C., Canada, 1982; Disjoint-interval topological sort: A useful concept in serializability theory, presented at "The 9th International Conf. on Very Large Data Bases, VLDB Endowment, Florence, Italy, October 1983."
16. T. IBARAKI AND T. KAMEDA, "Multi-version vs. Single-version Serializability," *Tech. Report* LCCR *83-1*, Lab. for Computer & Communications Research, Simon Fraser Univ., 1983.
17. J. L. W. KESSELS, The readers and writers problem avoided, *Inform. Process. Lett.* **10** (3) (1980), 159–162.
18. D. E. KNUTH, "The Art of Computer Programming, Vol. 3: Sorting and Searching," Addison–Wesley, Reading, Mass., 1973.
19. H. T. KUNG AND J. T. ROBINSON, On optimistic methods for concurrency control, *ACM Trans. Database Systems* **6** (2) (1981), 213–226.
20. S. MURO, T. MINOURA, AND T. KAMEDA, Multi-version concurrency control for a database system, CCNG Report No. E-98, Computer Communications Networks Group, University of Waterloo, Waterloo, Ont., Canada, August 1981.
21. C. H. PAPADIMITRIOU, P. A. BERNSTEIN, AND J. B. ROTHNIE, Some computational problems related to database concurrency control, *in* "Proceedings, Conf. Theoretical Comput. Sci., Univ. of Waterloo, Waterloo, Ont., Canada, August 1977," 275–282.
22. C. H. PAPADIMITRIOU, The serializability of concurrent database updates, *J. Assoc. Comput. Mach.* **26** (4) (1979), 631–653.

23. C. H. PAPADIMITRIOU AND P. C. KANNELLAKIS, On concurrency control by multiple versions, *in* "Proceedings ACM Sympos. Principles of Database Systems, March 1982," 76–82.
24. D. P. REED, Implementing atomic actions on decentralized data, *in* "Proceedings, 7th ACM Sympos. on Operating Systems Principles, December 1979," 66–74; *ACM Trans. Computer Systems* **1** (1) (1983), 3–23.
25. D. L. RUSSEL, State restoration in systems of communicating processes, *IEEE Trans. Software Engrg.* **SE-6** (2) (1980), 183–194.
26. R. E. STEARNS, P. M. LEWIS, II, AND D. J. ROSENKRANTZ, Concurrency control for database systems, *in* "Proceedings, IEEE Sympos. Found. Comput. Sci., October 1976," 19–32.
27. M. STONEBRAKER, E. WONG, AND P. KREPS, The design and implementation of INGRES, *ACM Trans. Database Systems* **1** (3) (1976), 189–222.