数理解析研究所講究録
第 482 巻 1983 年 183-214

MULTI-VERSION CONCURRENCY CONTROL SCHEME FOR A DATABASE SYSTEM — VERSION 1 —

Shojiro Muro (Kyoto University)
Tiko Kameda (Simon Fraser University)
Toshimi Minoura
(University of Southern California)

Abstract

We present a multi-version concurrency control scheme for a centralized database system, which allows increased concurrency. It grants an appropriate version to every read request without causing inconsistency. Transactions issuing write requests which would cause inconsistency are aborted. We state precisely when old versions can be discarded and describe in detail how to eliminate the effects of an aborted transaction. This paper is based on the notion of V-serializable execution which preserves consistency. We show that any "D-serializable" schedule of Papadimitriou (or "conflict-preserving" serializable schedule of Bernstein, et al.) is a special type of V-serializable execution and that even some non-serializable schedules can be turned into a V-serializable execution without changing the order of operations.

Presented at RIMS, Kyoto University, June 25, 1982

Muro Shojiro 室 章治郎
Kameda Tsunehiko 亀田 恒彦
Minoura Toshimi 箕浦 敏美

This paper is based on [MURO-82]. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant Nos. A5240 and A4315.
Authors's addresses: S. Muro, Dept. of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto 606, Japan; T. Kameda, Dept. of Computing Science, Simon Fraser University, Burnaby, B.C. V5A 1S6, Canada; T. Minoura, Dept. of Flectrical Engineering-Systems, University of Southern California, University Park, Los Angeles 90007, U.S.A.

1. Introduction

In a typical database system, many users access shared data concurrently. Unless some kind of discipline is imposed on user transactions, data in the system may be modified in an unintended way. The reader is referred to [GRAY-78] for anomalies that may occur.

A concurrency control mechanism should realize a high level of concurrency without causing inconsistency due to undesirable interactions among transactions. Many schemes for concurrency control have recently been proposed [BAYE-80, CASA-81, KUNG-81, PAPA-79, SCHL-78, STEA-76]. Earlier concurrency control schemes [ASTR-76, ESWA-76, STON-76] are mostly based on locking.

The idea of using multiple versions of objects in a database system was proposed in [STEA-76]. If we keep multiple versions of each object, there is more likelihood of being able to grant write requests that have arrived "too soon", since we can save older versions for future read requests. Consider the following schedule [ESWA-76]:

W1[X]R2[X]W2[Y]R1[Y],

where Ri[X] and Wi[X] respectively denote a read and write operation on object X by transaction Ti. Since T2 reads the value of X from T1 and T1 reads the value of Y from T2, the above schedule is not serializable [ESWA-76]. The problem here is that W2[Y] arrives too soon, i.e., before R1[Y]. If we keep the initial value of Y as well as the new value written by W2[Y], R1[Y] can access the old value, and the execution now becomes "serializable".

A family of multi-version concurrency control schemes were investigated in [STEA-76]. They also present a number of important concepts and results.

Bayer, et al. [BAYE-80] and Kessels [KESS-80] took notice of the fact that most database systems maintain two versions, the old version and the new version, of each object for recovery reasons while a transaction is modifying it. In the concurrency control scheme of [BAYE-80], a read request by a transaction is always granted without causing inconsistency. A read operation reads either the old or the new version, depending on the state of the object with respect to updating. This clearly increases concurrency.

Reed [REED-79] has also proposed a multi-version concurrency control scheme for distributed databases based on time stamping. This can of course be used in a centralized database system. However, it imposes a "serialization order" too early in the game, thus prohibiting some allowable executions.

Papadimitriou and Kanellakis [PAPA-82] examined the problem of concurrency control when the (centralized) database management system supports the multiple versions of data. They characterized the limit of parallelism achievable by the multiple versions approach and demonstrated the resulting space-parallelism tradeoff.

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. [STEA-76], 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 an unterminated transaction to be read by other transactions. This allows further concurrency. As in [PAPA-79] we allow each transaction to make at most one read access and at most one write access to each object, but unlike [PAPA-79] we allow read (and similarly write) operations to be performed at different times.

Section 3 introduces a useful tool called the history graph which represents the "history" of an execution. History graph is used in defining the V(version)-equivalence among executions. The V-serializability is then defined in Section4 based on this equivalence. The V-serializability is a generalization of "conflict-preserving" serializability [BERN-79] or "D-serializability" [PAPA-79] to the multi-version case.

In Section 5, we present our multi-version concurrency control algorithm (Algorithm MV) which schedules operations of user transactions by maintaining and updating relevant subgraphs of the history graph and dependency graph [ESWA-76]. One or more transactions are aborted if a write operation would create a cycle in the dependency graph. We prove that a read operation is always granted (i.e., there always is an appropriate version to be read without causing inconsistency). This is non-trivial, whereas a similar result in [CASA-81] (see also the "certification method" [BERN-81]) is more straightforward, since in his model a transaction consists of two steps, a read followed by a write, and atomic read of several objects cannot cause inconsistency. Finally in Section 6 we prove that Algorithm MV preserves consistency.

Our main contributions in this paper consist of the following.

1. Extension of the model in [STEA-76].
   A transaction can update an object without first reading it. We also allow versions written by an unterminated transaction to be read by other transactions.

2. Definition of V-serializability.

   This is an extension of the notion of "conflict-preserving" serializability [BERN-79] and "D-serializability" [PAPA-79] to the multi-version case.

3. Introduction of history graph.

   This graph faithfully records the history of execution, enabling partial back-outs. The graph can be embedded in the database itself, incurring little space overhead.

4. A concurrency control scheme based on the history graph and the dependency graph.

   We clearly indicate when transactions can be commited and old versions can be discarded. We also give a clear exposition of how to abort a transaction and how the effect of an abortion propagates, necessitating abortion of other transactions.

5. Discussion of "check-point" transactions [FISC-81].

Finally we comment that a practical multi-version database system based on time stamping has recently been built [DUBO-82], and experience with it indicates that keeping multiple versions can be practical in terms of time and space overhead.

This paper is a completely overhauled version (version 1) of our earlier report (version 0) [MURO-81].

2. Database System Model and Execution Sequence

In this section, we shall describe the database system model used in this paper. A database system consists of a set D of objets (or data items), a set T = {T1, T2, ..., Tn} of transactions, and a scheduler. A transaction starts with BEGIN and ends with TERMINATE. Other steps of a transaction are a sequence of read and write operations. A read operation Ri[X] of transaction Ti returns a value (or a version, see below) of object X, and write operation Wi[X] of transaction Ti creates a new value for X. Each object is accessed by at most one read and at most one write operation of each transaction. If a transaction Ti both reads and writes an object X, Ri[X] precedes Wi[X] in Ti, since Ti need not read what it has written.

A database state is an assignment of a value to each object in D. Some states are designated as consistent and the rest as inconsistent [ESWA-76]. The scheduler does not know which states are consistent, but it assumes that a transaction executed alone maps a consistent state into a consistent state. The initial database state is assumed to be consistent. We say that transactions execute concurrently if the operations of the transactions are interleaved.†

In order to avoid concurrent execution of transactions which renders the database state inconsistent, execution of some read

---

† We assume here that the execution intervals of the operations accessing the same objects do not overlap, but other operations may overlap in time.

and write operations may have to be delayed or rejected by the scheduler of the system. A concurrency control algorithm is the specification of a scheduler.

For each object X, its initial value is defined to be version 0, and a new version of X is created by each succeeding write operation accessing X. The version number of a value of object X created by write operation Wi[X] is denoted by #(Wi[X]), and is strictly larger than the version number of the previous version of X. The version number of a value read by read operation Ri[X] is denoted by #(Ri[X]). The version number will be discussed below in more detail.

Let OP(T) denote the set of all read and write operations of the transactions in T. An execution for T is defined as follows.

Definition 2.1. An execution for a set T of transactions is a triple (OP(T), <<, #), where << is a total order on the operations in OP(T) and # is a mapping from OP(T) to the set of nonnegative integers.

Let e = (OP(T), <<, #) be an execution. For two operations A and B in OP(T), we say that A precedes B in e if A<<B. Intuitively, << represents the time order among the operations in OP(T).† In the following we use two different representations for an execution. The linear representation, to be described in this section, is more convenient for representing the total order << clearly.

---

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

However, it does not represent complete information about the mapping #. The second representation, the history graph, introduced in the next section loses some information about the order <<. We leave a formal definition of the linear representation to the reader, and simply indicate it by an example.

Example 2.1. An execution (OP(T), <<, #) may be represented by the sequence

R1[X]R2[Z]W3[X]W1[Z]R3[Y]R3[Z]W4[X]R2[X]W3[Z]W2[Y].

The total order << is represented by the order (left to right) in which the operations of OP(T) appear. The arrow from W3[X] to R2[X], for instance, indicates that #(W3[X])=#(R2[X]), i.e., T2 reads the version of X written by T3. Note that the actual version numbers (which are not essential) are not shown in this representation. If there is no arrow leading to a read operation (such as R3[Y]), it indicates the read operation reads the initial version (i.e., version 0). □

For stating some definitions and theorems, it is convenient to introduce the fictitious "initailizing transaction", T0. It is a write-only transaction which writes version 0 of each object. All (write) operations of T0 precedes all operations of the other transactions and we do not explicitly show these write operations. We define #(W0[X])=0 for all objects X.

Given an execution (OP(T), <<, #), the following rules govern the relationship between the mapping # and the order <<.

V1. If Wi[X]<<Wj[X] then #(Wi[X])<#(Wj[X]).

V2. For each read operation Ri[X], there exists a

write operation Wj[X] such that Wj[X]<<Ri[X]
and #(Wj[X])=#(Ri[X]).

Informally, rule V1 says that the version number be incremented each time an object value is updated. Note that in rule V2 above, there is at least one write operation, i.e., W0[X], that precedes Ri[X]. Rule V2 implies that a read operation accessing an object must return the version created by a write operation preceding that read operation. It follows therefore that if Ri[X]<<Wj[X] then #(Ri[X])<#(Wj[X]). Note that more than one read operation may access the same version.

Version numbers are important in so much as they define a total order for the versions of each object and provide index for referring to different version. Therefore without loss of generality we may assume that they are consecutive integers starting at 0. This is automatically accomplished by referring to the jth version of an object, where j = 0,1,2, ...

There are certain executions with a pleasing property that the version numbers are implicitly defined by the order <<.

Definition 2.2. An execution (OP(T), <<, #) is said to be normalized if

a) the version numbers of each object are consecutive integers, and

b) if Wi[X]<<Rj[X] then #(Wi[X])≤#(Rj[X]).

Condition b) above requires, intuitively, that each read operation in a normalized execution read the "most recent" version of the object it accesses.

A normalized execution can be interpreted as an execution in the conventional "single-version" database. In fact the linear representation is a conventional way of representing such an execution [BERN-79, PAPA-79]. Arrows are omitted because it is understood that each read operation reads the result of the "most recent" write.

3. History Graph and Version-Equivalence

In this section we introduce the second, graphical representation for an execution. This graph will play a key role in the subsequent discussion. Let e = (OP(T), <<, #) be an execution and let D(T) denote the set of objects accessed by the operations in OP(T). In order to represent e, we construct a labelled bipartite graph, HG(e) = (N,A), called the history graph, where N and A are the set of nodes and set of arcs, respectively. Intuitively, for each transaction Ti in T, HG(e) represents the versions accessed by Ti. N consists of the following nodes.

N1: For each transaction Ti in T, there is a node ("transaction node") with label Ti.

N2: For each pair [X,v] such that X is in D(T) and v ($\geq$0) is the version number of a version of X, there is a node ("version node") with label [X,v].

The arcs in A are given as follows, where an arc (a,b) is directed from node a to node b.

A1: ([X,v], Ti) $\in$ A iff #(Ri[X]) = v.

A2: (Ti, [X,v]) $\in$ A iff #(Wi[X]) = v.

As we mentioned in Section 2, we may assume without loss of

generality that for each object X, its version numbers are consecutive integers starting at 0. This we assume in the rest of this section.

Definition 3.1. For a set T of transactions let e and e' be two executions. e and e' are said to be V-equivalent, written $e \equiv e'$, if HG(e) = HG(e').

Suppose e = (OP(T), <<', #') is V-equivalent to e' = (OP(T), <<', #'). Then Ti writes the jth version of an object X in e, iff Ti writes the jth version of X in e'. Similarly, Ti reads the jth version of X in e, iff Ti reads the jth version of X in e'.

Lemma 3.1. Let e' = (OP(T), <<', #') be an execution for a set T of transactions such that the version numbers for each object are consecutive integers. There exists a normalized execution e = (OP(T), <<, #) for T such that e and e' are V-equivalent.

Proof: Start with e = e', i.e., << = <<' and # = #'. If read operation Ri[X] in e does not access the version generated by the immediately preceding write operation of the form Wj[X] (Wj[X]<< Ri[X]), then bring Ri[X] forward in the linear representation of e to just before Wj[X], i.e., let Ri[X]<<Wj[X]. Repeat this process until no such Ri[X] exists for any i or X. The resulting sequence represents the desired normalized execution e. Clearly we have HG(e) = HG(e').

Q.E.D.

The above lemma provides an important link between the V-serializability introduced in the next section and "conflict-preserving" serializability [BERN-79] or "D-serializability" [PAPA-79]. This point will be discussed later in more detail.

The V-equivalence of two executions can be determined efficiently. For each i we can check the nodes adjacent to Ti in HG(e) and HG(e') in $O(|D(T)|)^{\dagger}$ time, because Ti has at most one read and one write operation accessing each object. Hence the V-equivalence of executions can be tested in $O(|D(T)||T|)$ time.

4. Serializability and V-serializability

Serializability [ESWA-76] of an execution is widely accepted as a useful notion. However, Papadimitriou [PAPA-79] has shown that serializability test is NP-complete [GARE-79], even for his simple transaction model, which implies that it is impractical to employ it in the actual implementation. Sufficient conditions for serializability which can be tested in polynomial time have been proposed [BERN-79, ESWA-76, KELL-75, PAPA-77, PAPA-79]. In this section we extend some of these ideas to the multi-version environment.

Definition 4.1. A normalized execution e=(OP(T), <<, #) is called serial if there is a total order <<< on the set T such that for any two distinct transactions T1 and T2 in T, if A<<B then T1<<<T2, where A (B) is any operation of T1 (T2).

In the linear representation of a serial execution, all operations of each transaction appear consecutively, and there is an arrow to each read operation from the nearest (on the left) write operation accessing the same object.

Definition 4.2. An execution e = (OP(T), <<, #) is said to be V-serializable if there is a serial execution e' = (OP(T), <<', #') such that $e \equiv e'$.

---

† For a set A, |A| denotes the cardinality of A.

Remark. As mentioned in Section 2, a normalized execution can be interpreted as an execution for a conventional or "single-version" database. In this connection, it turns out that every "D-serializable" schedule [PAPA-79] or "conflict-preserving" serializable schedule [BERN-79] is a normalized V-serializable execution.† However, the converse is not true, since our transaction model is more general than theirs in that we allow a read operation to follow a write operation in a transaction.

This observation has the following important implication. Consider a V-serializable execution e that is not "D-serializable" (see Section 1 for an example of such an execution). If e is given to a "D-serializing" scheduler or the "CPSR-scheduler" [CASA-81], some operations will be either delayed or rejected. However, a "V-serializing" scheduler would accept all operations of e without delay or rejection, which means increased concurrency. □

We now davelop a method for testing V-serializability of an execution. For a set of transactions T = {T1, T2, ..., Tn}, let e = (OP(T), <<, #) be a given execution. We construct for e a directed graph DG(e)=(T,B), called the dependency graph [ESWA-76]. The set of arcs of DG(e), B, is defined as follows:

(Ti, Tj) $\in$ B iff any of the following conditions holds for some X $\in$ D(T).

1. (ww-conflict) There exist two operations Wi[x] and Wj[X] in OP(T) such that #(Wj[X]) is the next larger version number after #(Wi[X]) (or #(Wj[X] = #(Wi[X])+1, if the version numbers for each object are consecutive).

---

† We assume here that a read (write) operation of the model in [PAPA-79, BERN-79] is changing into a sequence of consecutive read (write) operations, one for each object in the "read (write) set."

2. (rw-conflict) there exist two operations Ri[X] and Wj[X] in OP(T) such that #(Wj[X]) is the next larger version number after #(Ri[X]) (or #(Wj[X]) = #(Ri[X])+1, if the version numbers for each object are consecutive),
3. (Dependence) there exist two operations Wi[X] and Rj[X] in OP(T) such that #(Wi[X]) = #(Rj[X]).

We call arc (Ti, Tj) $\in$ B a primary arc if codition 3 above holds. Other arcs are called secondary arcs. Now let e be a serial execution as defined in Definition 4.1, and let Ti<<<Tj for Ti, Tj $\in$ T. Because of the rules V1 and V2 in Section 2, an arc between Ti and Tj in DG(e), if any, must be directed from Ti to Tj. Recall that rule V2 implies if Ri[X]<<Wj[X] then #(Ri[X])< #(Wj[X]). It follows that DG(e) cannot have a cycle if e is serial.

Lemma 4.1. Let T = {T1, T2, ..., Tn}, and let e = (OP(T), <<, #) and e' = (OP(T), <<', #') be two executions. If HG(e) = HG(e'), then DG(e) = DG(e').

Proof: If HG(e) = HG(e'), then each operation has the same version number in both e and e'. It follows from the definition of the dependency graph that DG(e) and DG(e') have the same set of arcs (as well as the same set of nodes).

Q.E.D.

Lemma 4.1 is true even if the version numbers for each object are not consecutive integers. We now prove an important theorem.

Theorem 4.1. An execution e is V-serializable if and only if DG(e) is acyclic.

Proof: First assume that e is V-serializable. Then by definition, there is a serial execution e' such that e $\equiv$ e'. By Defi-

nition 3.1, we have HG(e) = HG(e') and thus DG(e) = DG(e') by Lemma 4.1. Since e' is a serial execution, DG(e) = DG(e') is acyclic.

In order to prove the "if part" of the theorem, we assume that DG(e) is acyclic. Let Tp(1)<<<Tp(2)<<<... <<<Tp(n) be a total order obtained by topologically sorting [KNUT-73] the set {T1, T2, ..., Tn} of nodes of DG(e), where p(.) is a permutation. Note that if i<j then there is no path from Tp(j) to Tp(i) in DG(e). For each i ($1 \le i \le n$), let e(i) be the sequence of operations of Ti. Concatenate these sequences to construct the linear representation of a serial execution e', e(p(1))e(p(2))...e(p(n)). We claim that HG(e')= HG(e), therefore e $\equiv$ e' by Definition 3.1, and e is V-serializable.

To prove that HG(e) = HG(e'), note first that these graphs have the identical set of nodes (version nodes as well as transaction nodes). We now show that HG(e) and HG(e') have the same set of "write arcs". Let {T'1, T'2, ..., T'k} $\subset$ T be the set of all transactions that write X such that for i = 1, 2, ..., k, (T'i, [X,i]) is an arc in HG(e). Then for every i with $1 \le i \le k-1$ there is an arc from T'i to T'i+1 in DG(e). Therefore we must have T'1<<<T'2<<<... <<<T'k as a result of topological sorting and thus (T'i, [X,i]) is also an arc in HG(e').

Next, in order to prove that HG(e) and HG(e') have the same set of "read arcs", let T'i, i = 1, 2, ..., k, be as defined above. Suppose in e Tj($\neq$T'i+1) reads the ith version of X, i.e., ([X,i], Tj) is a "read arc" in HG(e). Then there are arcs from T'i to Tj and Tj to T'i+1 in DG(e), and therefore we have T'i<<<Tj<<<T'i+1. It follows that in e' Tj reads version i of X, i.e., ([X,i], Tj)

is also an arc in HG(e'). The case Tj = T'i+1 is easy to prove.

Q.E.D.

Corollary 4.1. For a given execution e = (OP(T), <<, #), the V-serializability of e can be checked in O(|D(T)||T|) time.

Proof: For each object X ∈ D(T), there are at most 2|T| read and write operations which access X, according to the definition of a transaction. Construct from the linear representation of e the sequence of operations accessing each object X in D(T). Such a sequence has at most 2|T| operations. We now renumber the version numbers of X by consecutive integers and construct DG(e) as follows. First intrduce the set T of nodes. For the sequence obtained for each object X, proceed as follows. Introduce an arc (Ti, Tj) if any of the following three conditions is satisfied (cf. the definition of the dependency graph).

1. There are operations Wi[X] and Wj[X] with
   #(Wj[X])=#(Wi[X])+1,
2. there are operations Ri[X] and Wj[X] with
   #(Wj[X])=#(Ri[X])+1,
3. there are operations Wi[X] and Rj[X] with
   #(Wi[X])=#(Rj[X]).

The sequence associated with each object can be processed in O(|T|) time, which implies that at most O(|T|) arcs are introduced per object. Therefore the totl number of arcs introduced is bounded by O(|D(T)||T|). A cycle in DG(e) can be tested in time linear in the number of nodes and arcs [AHOH-74].

Q.E.D.

5. The Algorithm

In this section, we discuss the multi-version concurrency control algorithm (Algorithm MV, for short) used by the scheduler of our database system. The input to the scheduler is the sequence of arriving requests from user transactions, including their BEGIN and TERMINATE requests. The BEGIN and TERMINATE requests from transaction Ti are denoted by b(Ti) and t(Ti), respectively.

5.1 General description

In response to each input request, Algorithm MV 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 dependency graph, which are maintained by the scheduler. They are initialized as follows:

$N \leftarrow \{[X,0] \mid X \in D\}$, $A \leftarrow \phi$,

$N' \leftarrow \phi$, $A' \leftarrow \phi$.

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

1. For BEGIN request, b(Ti):

   Create a transaction node Ti.

2. For Wi[X]:

   Create a version node [X,v], where v is one plus the currently largest version number of object X, and add an arc (Ti, [X,v]) to A.

3. For Ri[X]:

   Create an arc ([X,v], Ti) $\in$ 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(Ti):

   Delete node Ti and possibly one version node (per object) together with the arcs incident on them, if Ti satisflies the "deletion condition" (see below).

As we will see later a write request is not always granted. If Ti is aborted as a result of the rejection of Wi[X], the changes made in response to Wi[X] above must be undone. Furthermore, other operations and/or transactions may have to be backed out as a result, as we discuss in more detail below.

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

There are three points still left uncleaur at this point. They are

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 Ri[X]?

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

5.2. Rejection of write requests and abortion of transactions

When Wi[X] is received, both HG* and DG* are updated as described above. If a cycle is created in DG* as a result, then the partial execution generated so far is not V-serializable if Wi[X] is appended to it. That is to say, even if all currently untermi-

nated transactions immediately send TERMINATE requests, the generated execution is not V-serializable. We reject Wi[X], abort Ti, and undo the changes made to HG* and DG* in response to all the requests made by Ti. Note that as a result, the version numbers of some objects may become non-consecutive. Furthermore, all transactions corresponding to the nodes of DG* reachable from Ti by primary arcs must be aborted, since they have read the versions to be discarded. This aborting process might propagate one after another. This phenomenon is similar to the domino effect [RUSS-80] or cascading [BAYE-80]. In our scheme, this is the price we pay for increased concurrency.

Remark. 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 other transaction(s) than the Ti which issued the "offending" write Wi[X]. Making Wi[X] wait is a possible option only if other transactions are aborted [STEA-76]. Otherwise, a cycle will be created by Wi[X] no matter how long Wi[X] waits. This is the reason why we abort Ti. However, if the aborted transaction is immediately restarted, the so-called cyclic restart [STEA-76] may occur. For the discussion of cyclic restart, including methods to cope with it the reader is referred to [STEA-76]. Note also that a complete abortion may not be necessary, but a partial backing up may suffice. However, we use abortion for ease of exposition. All information needed for this purpose 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 back out for this purpose should also be taken into

account. □

5.3. Commit and deletion conditions.

Let $\widetilde{DG}$ be a partial graph of DG*, obtained from DG* by deleting all secondary arcs from it. Since we keep DG* acyclic, $\widetilde{DG}$ is *a fortiori* acyclic. If there is a path from a node to another node b in an acyclic directed graph, we call a a predecessor of b.

Definition 5.1. Transaction Ti is said to satisfy the commit condition if t(Ti) has been received by the scheduler and all predecessors of node Ti in $\widetilde{DG}$ have been committed.

When a transaction satisfies the commit condition, the system notifies the fact to the transaction. If we keep a record of information about all input operations forever, the total storage space needed by the algorithm will grow out of bound. 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 [STEA-76]. We say that a node of a directed graph is a source if it has no incoming arc.

Definition 5.2. Transaction Ti is said to satisfy the deletion condition if node Ti is a source in DG* and t(Ti) has been received by the scheduler.

A transaction satisfying the commit condition may also satisfy the deletion condition. If Ti 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.)

C1: Delete node Ti from HG* together with the arcs incident on it (both incoming and outgoing arcs).

C2: For each $X \in D$ such that Ti has written a version of the

form [X,v], delete the version node [X,v'] with v'< v.

C3: Delete node Ti from DG* together with its outgoing arcs.

Suppose that among the open transactions, T'1, T'2, ..., T'k have written versions of X, [X,v1], [X,v2], ..., [X,vk], where v1 < v2 <...< vk. Because of ww-conflict, there is an arc (T'i, T'i+1) in DG* for i = 1, 2, ..., k-1. Therefore they can satisfy the deletion condition only in the order, T'1, T'2, ..., T'k. If there is a open transaction T'0 which read a previous version [X,v0] with v0 < v1, then there is an arc (T'0, T'1) in DG* due to rw-coflict. T'1 can satisfy the deletion condition only after T'0 has been deleted. Rule C2 makes sure that, for i = 1, 2, ..., k, when T'i satisfies the deletion condition, the version [X,vi-1] is deleted. Each time C2 is applied, exactly one version (per object that Ti has written) is deleted. It is seen that versions are deleted in the order they were created (unless they are deleted with aborted transactions), and that exaclty 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 ϵ D, the only version node of the form [X,v] with no incoming arc in HG* is either of version 0 or the version with the largest version number among those written by the transactions which have been deleted.

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

There are two possible ways in which a transcation newly satisfies the deletion condition.

a) The scheduler receives a TERMINATE request, t(Ti).

b) A predecessor transaction of Ti in DG* has been deleted either by rule C2 or by abortion, making Ti a source in DG*.

In our algorithm, we delete the transactions satisfying the deletion condition whenever they are found.

5.4. Processing read requests.

If a read operation Ri[X] is accepted by the scheduler and accesses a version v of X, HG* and DG* must be updated. Arc ([X,v], Ti) is added to HG* and arcs due to rw-conflict and/or dependence are added to DG* . Unlike the "single-version" database system, the scheduler has a choice as to which version of X to allow Ri[X] to access. The following theorem states that we can always grant a read request.

Theorem 5.1. When a read request Ri[X] arrives, there always exists a version such that no cycle is created in DG* if the version is read by Ri[X].

Proof. Consider HG* and DG* just before the read request Ri[X] is received by the scheduler. If there is no open transaction that has written a version of object X, then Ri[X] can access version v, where [X,v] is the unique node in HG* stated in Lemma 5.1, because no new arc is created in the updated DG*.

Assume now that there are k ($\geq$ 1) transactions T'1, T'2, ..., T'k in HG* each of which has written a version of X. We denote the nodes created by these transactions by [X, v1], [X,v2], ..., [X,vk], where v1 < v2 <...< vk. Note that v0 < v1, where [X,v0]

is the unique version node without any incoming arc (see Lemma 5.1). Then there are arcs (T'1, T'2), (T'2, T'3), ..., (T'k-1, T'k) in DG*. If there is no (directed) path from Ti to T'j for any j ($1 \leq j \leq k$), Ri[X] can access version vk, because no cycle is created after adding arc (T'k, Ti) to DG*. Next, if there is a path from Ti to some T'j ($1 \leq j \leq k$), then let h be the smallest index ($1 \leq h \leq k$) such that there is a directed path from Ti to T'h. Thus if Ri[X] accesses the version vh-1, then arcs (T'h-1, Ti) and (Ti, T'h) are added to DG*. However, the former arc cannot introduce a cycle since there is no path from Ti to T'h-1, and the latter arc cannot introduce a cycle, since there was always a path from Ti to T'h. Consequently no cycle is created in the updated DG*.

Q.E.D.

Theorem 5.1. is an "existence theorem" and does not state how to find an appropriate version to grant to a read request. The most logical thing to do would be to look for the newest version that does not create a cycle by granting it. An open problem is how to efficiently determine that version.

5.5. Implementation

Here we discuss only the implemetation of HG* which facilitates its updating. The implementation of DG* is straightforward. For each object $X \in D$, we maintain a list of lists, both doubly linked, named VLIST(X) (see Fig. 1). Each list in VLIST(X) is headed by a record with four fields: version-number; value; written-by; 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 pointer to the linked list of all transactions that have read this version. Note that dependence exists between the transaction in the third field and those in the linked list. Similarly, ww-

conflict exists between two transactionns which wrote adjacent versions. And finally, rw-conflict 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 $X \in D$ provide all information about HG*, it is convenient to maintain another set of lists, two for each open transaction. The first list, WROTE(Ti), contains the pointers to all versions Ti has created and the second list, READ(Ti), contains the pointers to all versions Ti has read. Actually a pointer in READ(Ti) does not point to a version itself, but rather to the element Ti, which is in the list of transactions which read the version.

We illustrate how the deletion of a transaction Ti can efficiently be carried out using these data structures. First, for each element (pointer) in READ(Ti), delete the transaction pointed to by it. Next, for each element (pointer) in WROTE(Ti), delete the version pointed to by it. Dependence, ww-conflict, and rw-conflict involving Ti must be reexamined to perform the corresponding updates on DG*. Note that the abortion of one transaction may necessitate abortion of other transactions due to the domino effect. Each of the transactions reachable in DG* from an aborted transaction should also be aborted as above.

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 $\widetilde{DG}$ 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 $\widetilde{DG}$. Such a transaction can be used as a "check-point transaction" [FISC-81] 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 all read operations and all write operations of a transaction be each atomic and the write operations follow the read operations in each transaction. Then we have a model similar to the one used in [BERN-79, CASA-81, PAPA-79]. For this model, we can show that each read (=atomic set of read operations) can always be granted and the implementation described in subsection 5.5 compares favorably with that given in [CASA-81].

6. Correctness of the Algorithm

In this section we shall prove that Algorithm MV generates only V-serializable executions. Our approach is to make use of Theorem 4.1 for an arbitrary execution e generated by Algorithm MV and show that the dependency graph DG(e) 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 there were not deleted.

Let {T1, T2, ..., Tn} 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. Without loss of generality let T1, T2, ..., Tm, where $m \leq n$, be the transactions which satisfied the deletion condition up to the present

time. In the current DG* we thus have only the nodes corresponding to the open transactions, Tm+1, Tm+2 , ..., Tn. These transactions are still in the system either because they have not terminated or because they are not a source in DG*. The following lemma shows that if the dependency graph is constructed for all of T1, ..., Tn, it will be acyclic.

Lemma 6.1. Let {T1, T2, ..., Tn} 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: Let T1, ..., Tm be the transactions which 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 Tm back to it. We want to show that the resulting graph is also acyclic. Since Tm 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 Tm 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.

Q.E.D.

It is now easy to prove a main theorem of this paper.

Theorem 6.1. Any execution allowed by Algorithm MV is V-serializable.

Proof: Let T1, T2, ..., Tn and the index m be as defined just before Lemma 6.1. To prove the theorem, we first consider the case where m = n, i.e., all transactions are completed. It follows from Theorem 4.1 and Lemma6.1 that the resultant execution is V-serializable.

Consider now the situation where there are open transactions which may be aborted in the future, i.e., the case where m < n. We claim that no matter what happens to them in the future, the partial execution consisting of the operations of T1, ..., Tm is V-serializable. This is clear since that subgraph of the dependency graph which is defined by the nodes T1, ..., Tm is acyclic and future actions of the scheduler will never create an arc directed to any of T1, ..., Tm, hence the subgraph in question will remain acyclic. In fact, this subgraph will never change in the future.

Q.E.D.

7. Conclusion

We have proposed a concurrency control scheme which maintains multiple versions of each data object. This avoids inconsistency to be caused by a read operation, since an appropriate version to be read is always available. It also enables more concurrency by allowing "write-ahead." If it becomes apparent that a write operation must be rejected in order to avoid inconsistency, we abort not only the transaction that issued the write request but also others that have read the versions written by the aborted transactions. It is possible to salvage some computation of a transaction to be aborted up to the first read operation that read an invalidated version, by referring to the partial history graph HG*.

The secondary arcs of the dependency graph were introduced more for expediency than for necessity. For example, if Wj[X] arrives after Wi[X], we let the Wj[X] create a newer version (i.e., version with a larger version number) than Wi[X]. However, it might be possible to let Wj[X] create an "older" version than Wi[X] in order to avoid a cycle that would otherwise be created. This possibility exists only if Tj did not read X. We intend to explore such possibilities as an extension of the work reported in this paper.

As stated in Introduction, Stearns et al. [STEA-76] 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 may increase the possibity of propagating domino effects. More quantitative analysis is required to see if the removal of this restriction is justified.

Acknowledgment

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 and T. Ibaraki of Kyoto University for their support and encouragement. He also acknowledges stimulating discussions with Hide Tokuda of the C.C.N.G. at the University of Waterloo. Abdul Farrag of Simon Fraser University has also provided us with useful comments.

References

[AHOH-74] Aho, A.V., Hopcroft, J.E., and Ullman, J.D.,"The Design and Analysis of Computer Algorithms", Addison-Wesley Reading, Mass., 1974.

[ASTR-76] Astrahan, M.M., et al., "System R: Relational approach to database management", ACM TODS 1, 2 (June 1976), 97-137.

[BADA-80] Badal, D.Z., "The analysis of the effects of concurrency control on distributed database system performance", Proc. 6th Intn'l Conf. on VLDB (Oct. 1980), 376-383.

[BAYE-80] Bayer, R., Heller, H., and Reiser, A., " Parallelism and recovery in database systems", ACM TODS 5, 2 (June 1980), 139-156.

[BERN-79] Bernstein, P.A., Shipman, D., and Wong, W., "Formal aspects of serializability in database concurrency control", IEEE Trans. Software Eng. SE-5, 3 (May 1979), 203-216.

[BERN-81] Bernstein, P.A. and Goodman, N., "Concurrency control in distributed database systems", ACM Comp. Surveys 13, 2 (June 1981), 185-222.

[CASA-81] Casanova, M.A., "The concurrency control problem for database systems", In Lecture Notes in Computer Science 116, Springer Verlag, 1981.

[DUBO-82] DuBourdieu, D.J., "Implementation of distributed transactions", Proc. The 6th Berkeley Workshop on Dist. Data Manag. and Comp. Networks, (Feb. 1982), 81-94.

[ESWA-76] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The notions of consistency and predicate locks in a database system", CACM 19, 11 (Nov. 1976) 624-633.

[FISC-81] Fischer, M.J., Griffeth, N.D., and Lynch, N.A., "Global states of a ditributed system", Proc. IEEE Symp. on Reliability in Distributed Software and Database Systems, (July 1981), 33-38.

[GARE-79] Garey, M.R. and Johnson, D.S., "Computers and Intractability- A guide to the theory of NP-completeness", Freeman, 1979.

[GRAY-78] Gray, J.N., "Notes on data base operating systems", In Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 393-481.

[KELL-75] Keller, R.M., "Look-ahead processors", ACM Computing Surveys 7, 4 (Dec. 1975), 177-195.

[KESS-80] Kessels, J.L.W., "The readers and writers problem avoided", Info. Process. Letts. 10, 3 (April 1980), 159-162.

[KNUT-73] Knuth, D.E., "The Art of Computer Programming, Vol.3: Sorting and Searching", Addison-Wesley, Reading, Mass., 1973.

[KUNG-81] Kung, H.T. and Robinson, J.T., "On optimistic methods for concurrency control", ACM TODS 6, 2 (June 1981), 213-227.

[MINO-80] Minoura, T., "Resilient extended true-copy token algorithm for distributed database systems", Ph.D. Thesis, Stanford University, May 1980.

[MURO-81] Muro, S., Minoura, T., and Kameda, T., "Multi-version concurrency control for a database system", CCNG Report E-98, Computer Communications Networks Group, University of Waterloo, August 1981.

[MURO-82] Muro, S., Minoura, T., and Kameda, T., "Multi-version concurrency control scheme for a database system", Technical Report TR 82-2, Dept. of Computing Science, Simon Fraser University, February 1982.

[PAPA-77] Papadimitriou, C.H., Bernstein, P.A., Rothnie, J., "Some computational problems in database concurrency control", Proc. Conf. Theoretical Comp. Sci., Univ. of Waterloo, (Aug 1977), 275-282.

[PAPA-79] Papadimitriou, C.H., "The serializability of concurrent database updates", JACM 26, 4 (Oct. 1979), 631-653.

[PAPA-82] Papadimitriou, C.H. and Kanellakis, P.C., "On concurrency control by multiple versions", Proc. ACM Symp. on Principles of Database Systems, (March 1982), 76-82.

[REED-79] Reed, D.P., "Implementing atomic actions on decentralized data", Proc. 7th ACM Symp. on Operating Systems Principles, (Dec. 1979), 66-74.

[RUSS-80] Russel, D.L., "State restoration in systems of communicating processes", IEEE Trans. Software Eng. SE-6, 2 (March 1980), 183-194.

[STEA-76] Stearns, R., Lewis, P., and Rosenkrantz, D., "Concurrency control for database systems", Proc. IEEE Symp. Foundations of Comp. Sci. (Oct. 1976), 19-32.

[STON-76] Stonebraker, M., Wong, E., and Kreps, P., "The design and implementation of INGRES", ACM TODS 1, 3 (Sept. 1976) 189-222.

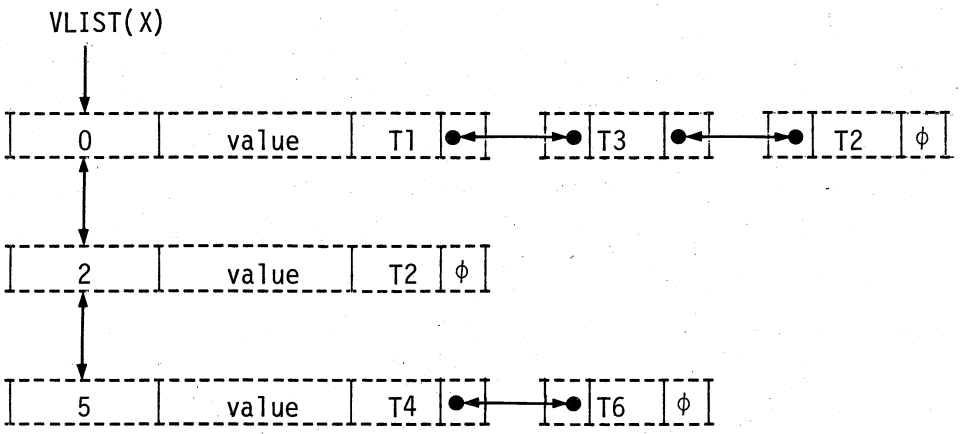

Fig. 1. Data Structure for Implementing HG*.