数理解析研究所講究録
第547巻 1985年 79-111

MULTIVERSION CONCURRENCY CONTROL SCHEME FOR A DISTRIBUTED DATABASE SYSTEM

Shojiro Muro
室 章治郎
Kyoto Univ.
京都大学

Tadashi Mizutani
水谷 匡
IBM Japan
日本IBM(株)

Toshiharu Hasegawa
長谷川 利治
Kyoto Univ.
京都大学

Abstract

A new multiversion concurrency control scheme for a distributed database system is proposed in this paper. In our database model, two versions (not copies) of each data object are allocated in different sites in the system. Different from usual distributed database systems with redundant copies, these two versions are not concurrently updated, but only one of them is updated for an update request. Since two versions are accessible for each read request and concurrent updates of two versions are not necessary, our scheme allows increased concurrency. The control algorithm is based on both timestamp mechanism and locking mechanism with two different modes, and it grants a version for every read request without causing inconsistency. Transactions with write requests which would cause inconsistency are aborted. It is proved that the new concurrency control scheme works correctly; namely, it preserves consistency without any deadlock and livelock among operations of the transactions in the system.

S. Muro and T. Hasegawa are with the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto 606, Japan.

T. Mizutani was with the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto 606, Japan. He is now with IBM Japan, Ltd.

1. Introduction

In usual database systems (abbreviated DBS), many users access shared data concurrently. Unless some kind of discipline is imposed on users' transactions, data in the system may be modified in an unintended way. Concurrency control is the activity of synchronizing read and write operations issued by concurrently executed transactions on a shared database and to realize a high level of concurrency without causing inconsistency due to undesirable interactions among transactions [ESWA-76]. Particularly, the purpose of concurrency control is to produce an execution of operations with the same effect as a serial (noninterleaved) one. Such executions are called serializable (see, e.g., [PAPA-79], [IBAR-83], [BRZO-84]). Many schemes for concurrency control, most of which are based on serializability theory, have been proposed in this decade for both of centralized DBS and distributed database system (abbreviated DDBS) models (e.g., see the references of the paper [BERN-81]).

Recently, multiversion concurrency control schemes have been investigated in many papers to achieve serializability by supporting multiple versions of the data objects (see, e.g., [REED-78], [BAYE-80a], [BAYE-80b], [STEA-81], [BERN-83], [PAPA-84], [MURO-84]). The main idea of the multiversion concurrency control scheme is as follows:

Each write operation on a data object, say, X does not overwrite the old value of X by the new one, but to produces a new version of X. If subsequently another transaction reads X, it selects one of the versions of X to be read. Since write operations do not overwrite each other, and since read operations can read any version, a high level of parallelisms is achieved in controlling the order of read and write operations in spite of additional bookkeeping.

Different from centralized DBS's, concurrency control for DDBS's is in a state of extreme turbulence. Although many concurrency control schemes have been proposed for DDBS s, they are usually complex and hard to understand (see the survey paper [BERN-81]). As what makes the problem hard, we list up the following two factors:

(F1) Users may concurrently access data object stored in many different sites in a system.

(F2) A concurrency control mechanism at one site cannot instantaneously know about interactions at other site, which mainly comes from the communication delay between sites.

One of the most critical consistency problems of DDBS's is concurrent writing for multiple (redundant) copies of a data object, which is caused by (F2).

Multiversion concurrency control schemes have already been proposed for an environment with distributed systems (e.g., [REED-78], [BAYE-80b], [STEA-81], [BERN-83], [PAPA-84]). However, in the models of these papers, versions of each object are not allocated in different sites but considered for each object in a site.

In this paper, we propose a multiversion concurrency control algorithm for a new type of DDBS's (we call this algorithm **DMV**), where not the copies but the versions of each data object are allocated in different sites in the system. In particular, as the first step to consider such model, the most simple model is examined; namely, there are two versions (the new one and the old one) for each data object and they are allocated at different sites (note that, in case of updating, there exist three versions temporarily). A read (write) request from a site first access to the nearer site with a version of the object from (to) which it reads (writes), respectively, where "nearer" means shorter distance, cheaper communication cost, and so forth. If the request can not be granted at the nearer site, then it is sent to the other site with another version of that object. The advantages of our model are as follows:

(A1) High reliability.

Each object is located at two different sites.

(A2) Rapid access to data object.

Read and write requests first access to the nearer site, and may be granted at that site.

(A3) Increased concurrency.

Since there are two (or temporally three versions) for each object, read requests which have arrived "too late" from the view point of serializability can also be granted. In fact, read requests are never rejected in our scheme.

(A4) Simple updating.

We can avoid complicated concurrent updating of redundant copies of an object by updating a version of the object.

(A1) is due to that versions are distributed and this is also a typical advantage of DDBS's. (A3) is obtained from the idea of concurrency control mechanism of multiversion stated above. (A4) as well as the advantages of write requests of (A2) are provided by introducing the new model of multiversion DDBS's in this paper, and these overcome the problem of concurrent updating redundant copies.

The concurrency control scheme for the proposed model is based on timestamp mechanism (see, e.g., [LAMP-78], [THOM-79]) and locking mechanism with two different modes which are based on a-lock and c-lock of [BAYE-80a], and it grants a version for every read request without causing inconsistency. Transactions issuing write requests which would cause inconsistency are aborted. It is proved that the new concurrency control scheme works correctly, i.e, it preserves consistency without any deadlock and livelock of operations of the transactions in the system.

2.The Model

In this section, we describe a DDBS model assumed in this paper.

2.1. Distribution of data objects

In our system model, data objects are distributed as follows:

1. All logical data objects are named X, Y, ..., and so on. These are distributed among the sites S_1, ..., S_N in a computer network, where we assume that there are N sites in a system.
2. Normally each logical data object X has two physical versions (see Remark 2.1); namely, old version X_O and new version X_N, and these are allocated to different two sites in the system. A site with X_N (X_O) is called the new site (old site) of X, respectively.

3. Each site has a <u>catalog</u> for all objects, which keeps the information con-cerning sites to which two versions of each object are allocated. Thus each site knows the <u>nearer site</u> and <u>farther site</u> of each object.
4. At each site S_i, a special process called <u>local controller</u> is equipped, which is responsible for the synchronization of the events in the system. It is main study of this paper to specify the algorithm executed by this controller, and this is described in section 4. A local controller consists of two components <u>transaction manager</u> (TM) and a <u>lock manager</u> (LM).
5. Any two local controllers can communicate via message exchanging. Assume that if a site sends more than two messages to another site, these messages are received by the destination site in the order that they are sent by the original site.
6. Transaction T issued from site S_i is controlled by the local controller of the S_i. Site S_i is called the <u>home site</u> of T. If transaction T requires a data object which is not at S_i, say, X, its request message is sent to the nearer site S_j with a version of X, and the local controller of S_j deals with this request according to the concurrency control algorithm. If the request message is granted at site S_j, then the acknowledging message is sent back to site S_i; otherwise, the request message is further sent to site S_k with a version of X (S_k is the farther site for S_i concerning the data object X).

Fig.1 shows an example of our DDBS model. Different from usual DDBS's with redundant copies, in our system, all the physical data objects of a logical data do not have the same data values and need not be updated concurrently. This overcomes the most complicated problem of the concurrency control in DDBS's, i.e., concurrent updating of redundant copies.

Remark 2.1. As will be discussed in section 4, every data object has temporarily three versions. One more version is called the newest version and denoted by X_{NN}. [ ]

2.2. Lock modes

Our concurrency control is mainly due to locking mechanisms. Transactions only use write-locks with two different modes called a-lock (analyze-lock) and c-lock (commit-lock). Idea of a-lock and c-lock was first introduced by Bayer, et al. [BAYE-80a] (see Remark 2.2). The a-lock is an exclusive write-lock for newest versions of objects. If a newest version of a data object, say, X_{NN} is a-locked, then neither read nor write operation can access X_{NN}. On the other hand, if X_{NN} is c-locked, any read operation can access X_{NN}, though no write operation can access X_{NN}. The c-lock is employed in this paper in order to keep the number of versions of eack object at most three.

When a write request for an object X is received by its home site, its a-lock on object X must be granted. If alock is granted, the newest version X_{NN} is created at old site of X, and then a-lock is coverted into c-lock mode. At this point, object X has three versions. When the local controller of the old site of X certifies that any read operation can be granted without reading the old version X_O, X_O is discarded and the LM of the controller releases the c-lock on X_{NN}. At the same time, the newest version X_{NN} (new version X_N) is renamed the new version X_N

(old version X_O), respectively. To acknowledge the timing to discard X_O, so called discard condition (see section 3) is examined by the local controller of the old site of X.

Remark 2.2 Another lock mode, r-lock(read-lock) was employed to grant read requests in [BAYE-80a]. In our scheme, every read request is granted without r-locks. [ ]

2.3. Transaction model

A transaction starts with BEGIN and ends with TERMINATE. Other steps of a transaction are a sequence of read and write operations. The read and write operation of transaction T_i to an object X are denoted by $R_i[X]$ and $W_i[X]$, respectively (subscripts are often omitted). Each object is accessed by at most one read and at most one write operation of each transaction. If a transaction T_i both reads and writes an object X, then $R_i[X]$ precedes $W_i[X]$ in T_i. This is by the assumption that T_i need not read what it has written. The read set (write set) of T is defined as the set of objects that T reads (writes), respectively.

Any write operation to an object must a-lock the object before it is executed. We assume that a transaction T sends the lock mode conversion messages to convert all of its a-locks on its write set into c-locks when T terminates its execution (i.e., T is committed). This is to avoid cascading [BAYE-80a] or domino effect [RUSS-80]. Since it is intricate to cope with cascading in decentralized systems, the newest version, say, X_{NN} is not accessible for any read operation until the controller of the site with X_{NN} receives the lock mode convertion message for X_{NN}. After the conversion, the LM of the controller of the site with the objects c-locked examines the discard condition of all the objects c-locked, and if the discard condition of an object is satisfied, then the c-lock on the newest version of this object is released. Consequently, transactions observe the 2PL (2-phase locking) mecha-nism if we don't distinguish between a-lock and c-lock. Fig.2 shows a configuration of execution of a transaction T. T is called active before it is committed, and T is called inactive after it is committed. A transaction is not aborted after it becomes inactive and never be backed up due to inconsistency and dead lock. Finally, if all of the operations of a transaction T are processed successfully within a finite time, we say that T is committed within a finite time after it is initiated.

Remark 2.3. The conversion of a-lock into c-lock corresponds to the release of usual exclusive write-lock in (r,x)-protocol [BAYE-82], since any read operation can read the newest version after the conversion. If we neglect the time delay for converting all a-locks into c-locks, the lock mechanism corresponds to strictly 2PL [BAYE-80a]. []

2.4. State transition of the object

Each object X has the following three states according to the writing rule concerning the object X described above:

D1. The normal state with two versions (X_O and X_N): X is neither a-locked nor c-locked, and any read request for X can read either X_O or X_N.

D2. X is a-locked by a transaction and the newest version X_{NN} is created: There are three versions of X, but X_{NN} is not accessible. Any read request for X can read either X_O or X_N.

D3. a-lock on X has been converted to c-lock: X_{NN} can be accessible to any read request. Any read request for X can read one of the three versions.

In each state, the version read by a read operation is decided by the concuurency control algorithm discussed later. Fig.3 shows the state transition diagram of object X.

A transaction can a-lock an object X in state D1. In state D2, if the update transaction is aborted, then X_{NN} is discarded and the state goes back to D1. In state D3, if the discard condition for the object X is satisfied, then X_O can be discarded, c-lock on X is released, and the state goes back to D1. When the state goes back from D3 to D1, X_{NN} (X_N) in state D3 becomes newly X_N (X_O) in state D1, respectively. Therefore, the old site and the new site of object X alternate with each other after the release of c-lock on X.

3. Consistency and Timestamp Mechanism

In this section, first, we give the notion of consistency, and then introduce the timestamp mechanism for achieving the consistency in our DDBS.

3.1. Consistency

If a transaction T is executed, then the user issuing transaction T can get a "view" of the database by its read operation. Also, the transaction T can change the state of the database by its write operation. A concurrency control algorithm must preserve consistency of a database (see, e.g., [ESWA-76], [ULLM-

80]). Under an assumption that each transaction keeps consistency of the database if it is executed independently, a role of concurrency control algorithm is to schedule operations of concurrently executing transactions such that (the database)/(the users) observe an execution with the same effect as some serial (noninterleaved) execution of the operations (see, e.g., [BRZO-84], [ROSE-84]). In this paper, the algorithm works so that both database and users observe an execution with the same effect as a serial one.

Definition 3.1. For a set of transactions which have been executed so far in the system under a concurrency control algorithm, the concurrency control algorithm is said to preserve consistency if the database and the users observe a execution of the operations issued by the transactions as if the transactions were executed sequentially. [ ]

Definition 3.2. A concurrency control algorithm is said to be correct if it preserves consistency and each transaction is either committed or aborted within a finite time after its initiation. [ ]

3.2. Timestamp mechanism

To develop a correct on-line concurency control algorithm, so called the dependency graph (abbreviated DG) are often used to represent dependency relation among the concurrently executing transactions concerning their execution order (see, e.g., [PAPA-79], [IBAR-83], [MURO-84]). Dependency relations are obtained by conflict relations of transactions' operations (see [PAPA-79], [IBAR-83] for the definition of "conflict"). Concurrency control algorithms employing the DG usually achieve their correctness by keeping the DG acyclic.

A concurrency control scheme based on the DG might be realized as follows in our database model: Suppose that a read or write request are received by their home sites in the system. If no conflict occurs with any request of other transactions, then that request is granted immediately; otherwise, a decision whether the controller grants the request or not is done by the following program:

```
add an arc(s) to DG.
if no cycle is caused
then < action G >
else < action R >
fi
```

In the program, action G (action R) means to grant (reject) the request, respectively. If action R is performed, arcs in DG are removed to reflect the rejection of the request.

In an environment of centralized DBS, updation of the DG is easy since one central controller can capture all events in the DBS easily and rapidly. However, in the DDBS's, it is hard for the local controller of each site to reflect in its DG all of the new information concerning the dependency of all transactions in the system, since it requires too many messages to be exchanged and too long message transmission delay. In [BAYE-80b] and [SUGI-84], algorithms to detect cycles by merging necessary information from distributed DG graphs are proposed, and several complicated situations occurred in these algorithms are revealed. Thus, in order to resolve the above complicated problems, we employ the timestamp mechanism to decide a serialization order of transactions in advance. When the BEGIN of each transaction is received by a local controller, a timestamp (TS) is provided from the local controller which is unique number in the entire network. This timestamp order gives a serialization order of the execution of transactions. That is, whenever two transactions T_i and T_j access one object, the access order must be consistent with the timestamp order of the two transactions; otherwise, backup is necessary. Thus, the following algorithm is applied to any pair of operations accessing the same object:

```
if TS(T_i) < TS(T_j)
then < action G >
else < action R >
fi
```

TS(*) denotes the timestamp of the transaction in the program. In this timestamp mechanism, by comparing two timestamps, the decision of execution order can be made easily and immediately without sending messages to other sites. As compared with the mechanism employing the DG, this rapidity is a great advantage in actual systems. However, concurrency decreases; namely, due to the total order among transactions imposed ahead of time, more transactions aborted unnecessarily. This is the price paid for simplicity and the reality of the concurrency control proposed in this paper (as for the importance of simple mechanism in DDBS's, see, e.g., [BERN-81]).

Concerning timestamp mechanism, we assume the following:

Assumptions on timestamps

1. Each transaction is assigned a timestamp at its home site when BEGIN of the transaction is received.
2. Each timestamp is unique.
3. At each site, values of timestamps are increasing, that is, if BEGIN of T_j is received after T_i at the same site, then

$TS(T_i)<TS(T_j)$ holds.

4. Once a timestamp is assigned, its TS is not updated nor deleted.
5. The timestamps of read and write operations of transaction T is the same as that of T.

The uniqueness of assumption 2 can be achieved as follows [THOM-79]. When BEGIN of a transaction is received by its home site, the TM of a local controller at the site assigns a timestamp to the transaction, where the timestamp consists of the higher bits (=the local clock time) and the lower bits (=unique TM identifier). The TM does't agree to assign another timestamp until the clock time proceeds to the next tick. Note that clocks at different sites are not required to set the same time. By these mechanisms, each timestamp is unique in the system, and the assumption 3 is also realized.

4. Concurrency Control

In this section, we describe the outline of our concurrency control algorithm DMV and the mechanism of TM and LM of a local controller to realize the algorithm DMV. Readers who are interested in the detailed algorithm, please refer to the appendix of [MIZU-84].

4.1. Outline of the algorithm DMV

The basic discipline of the algorithm DMV is to synchronize all of the events in the system to be consistent with the timestamp order. Fig.4 shows a snapshot of the state of an object X concerning timestamps. For convenience, the timestamps of the transactions which wrote X_O, X_N, and X_{NN} are denoted by TS(X_O), TS(X_N), and TS(X_{NN}), respectively. $TS(X_O)<TS(X_N)<TS(X_{NN})$ must be satisfied since the algorithm DMV synchonize all of the requests in the system in the timestamp order.

4.1.1. Execution of read request

First, we consider how a read request for X of a transaction T is executed by the algorithm DMV according to the state of object X, i.e., D1, D2, and D3.

[The algorithm in state D1]

Case 1. If $TS(X_O)<TS(T)<TS(X_N)$, then the request reads the value of X_O.

Case 2. If $TS(X_N)<TS(T)$, then the request reads the value of X_N.

Note that $TS(X_O)<TS(T)$ is guaranteed by the discard condition described later.

[The algorithm in state D2]

Case 1. If $TS(X_O)<TS(T)<TS(X_N)$, then the request reads the value of X_O.

Case 2. If $TS(X_N)<TS(T)<TS(X_{NN})$, then the the request reads the value of X_N.

Case 3. If $TS(X_{NN})<TS(T)$, then let the request to wait until the a-lock to the object X is converted to the c-lock or the transaction with a-locking is aborted.

If we let the read request satisfying $TS(X_{NN})<TS(T)$ read X_N, it forces the transaction a-locking to be backed up, since otherwise the execution order of T and the transaction a-locking results in the reverse of the timestamp order. Thus, in such a case, we let

the read request of transaction T wait until the state of object X becomes D1 or D3.

[The algorithm in state D3]

Case 1. If $TS(X_O)<TS(T)<TS(X_N)$, then the request reads the value of X_O.

Case 2. If $TS(X_N)<TS(T)<TS(X_{NN})$, then the request reads the value of X_N.

Case 3. If $TS(X_{NN})<TS(T)$, then the request reads the value of X_{NN}.

In the above algorithm, if the elected version of X for a read request is not in the site where the request first accessed, then this request is transmitted to the other site where the requesting version exists. To make these decisions immediately in distributed control, two sites with the same object inform with each other the version numbers of the object if the new version is created or aborted.

4.1.2. Execution of write request

Next, we describe how a write request of transaction T to object X is executed by the algorithm. Because of the locking mechanism, a write request is granted only if the state of object X is in D1.

[The algorithm in state D1]

Case 1. If $TS(X_N)<TS(T)$ and max-$R(X_N)<TS(T)$ hold, where max-$R(X_N)$ means the maximum timestamp of the transactions which have read X_N, then a-lock for object for object X is granted to transaction T, and T creates the

newest version X_{NN} with the version number TS(T).

Case 2. If the condition of case01 is not satisfied, the write request by T_{NN} is rejected.

Note that if max-R(X_N)>TS(X_N) holds if there exists a least one transaction which reads X_N.

Now, assume that a transaction T writes X_{NN} and the state of object X is in state D2 or D3. If another transaction T' with write request to X is generated in the system, then this request is coped with as follows:

[The algorithm in state D2]

Case 1. If TS(T)<TS(T'), then the request of T' waits for the release of a-lock from transaction T.

Case 2. If TS(T')<TS(T), then T' is aborted.

[The algorithm in state D3]

Case 1. If TS(T)<TS(T') and max-R(X_{NN})<TS(T'), then the request of T' waits for the release of a-lock from transaction T, where max-R(X_{NN}) is defined similarly to max-R(X_N).

Case 2. If the condition of case 1 is not satisfied, then T' is aborted.

In the algorithm DMV, when a transaction T releases its a-lock, a-lock is next granted to the transaction with the minimum timestamp among those currently waiting for the release of a-lock.

4.1.3. Discard condition

Finally, we discuss the discard condition. In state D3, if it is acknowl-dged that no read request with a timestamp less than TS(X_N) will be generated in in future, then c-lock of X_{NN} is released and the state of object X goes back to D1. Now, we shall make clear the condition necessary for the above acknowledgement. Assume that an active transaction T^* has the minimum timestamp among those which are active in the system. Then, define MINAC by MINAC=TS(T^*). From the assumptions on timestamps, the values of timestamps are monotone increasing with respect their issuing time. Thus, if TS(X_N)<MINAC is satisfied, there exists a version for any read request of active transactions in the system even if X_O is discarded. Thus, we call this condition TS(X_N)<MINAC the discard condition of object X.

4.2 Configuration of version

It is important to consider the physical configuration of a version for realizing the algorithm (see Fig.5). A version has a name of the object and a value. In addition, "writer", "readers", "max-reader", "relative version number" are also stored as parts of data of a version. "writer" is a timestamp of a transaction which wrote it, this timestamp value is the key index of each version. Similarly, "readers" are the timestamps of transactions which have read the version. "max-reader" is the maximum value of timestamps in "readers". While "readers" are empty, "max-reader" is set to the value equal to "writer". "relative version number" stores the relative version number among the versions of that object; namely, when the version is created, it is set to "NN", and each time the old version of that object is discarded, it is converted to "N", and then to "O".

4.3. Local controller

To realize the algorithm DMV, LM and TM of local controller C_i of site S_i use several buffers, logs, and flags. We shall describe their mechanisms.

4.3.1 Transaction manager(TM)

A TM of C_i has the active transaction list (ATL) which is a list of active transactions at S_i. The minimum value of timestamps of the transactions in the ATL at S_i is denoted by MIN_i. A TM of C_i has also transaction log's (TL's) which record the operations of the transactions whose home sites are S_i. For a transaction T_k with home site S_i, if a request of T_k is sent to another site, then this event is recorded in the TL of T_k together with the name of site to which the request is sent. When the reply for the request of T_k is sent back to S_i, this event is also recorded in the TL of T_k together with the name of site at which the version is read or written by T_k. In particular, for a read request, the writer of the version read by this request is also recorded. The TL is referred to when the transaction is aborted.

4.3.2. Lock manager(LM)

A LM of C_i keeps and updates the values of MIN_1, ..., MIN_N in a buffer. This buffer is called the minac board of site i and is denoted by MB_i. $MB_i(k)$ denotes the value of MIN_k in the MB_i. Suppose that at site S_j, the transaction with timestamp MIN_j becomes inactive or is aborted. Then the value of MIN_j is updated, and the new value of MIN_j is informed to all other sites. In case that no active transaction is at S_j, the C_j reads the local clock time and creates the timestamp which is set to the new value of MIN_j. C_j repeats this process periodically until an active transaction with home site S_j is generated. The minimum value of MB_i is denoted by $MINAC_i$. $MINAC_k$ (k=1, ..., N) has the same value in the network except for the time interval of message delays. Thus subscripts are often omitted so far as no confusion occurs. The MINAC is the minimum timestamp of the currently active transactions throughout the system.

For each object at S_i, a flag and buffers are maintained by the LM. The <u>mode flag</u> for X is denoted by MF(X) which exhibits the lock mode on X. That is, MF(X)=ϕ or a or c. The <u>information board</u> for X is a buffer and is denoted by IB_X which keeps the timestamps of the transactions that wrote the existing versions of X; namely, the IB_X consists of three indices. If an object X is in state D1, then the IB_X(NN) is empty. When X_O is discarded in state D3, the IB_X is shifted; namely, IB_X(N) and IB_X(O) are newly set to old IB_X(NN) and IB_X(N), respectively, and IB_X(NN) becomes empty. The information of these MF(X) and IB_X is for dealing with read and write requests for X. Assume that X is located at two sites S_i and S_j. Then, MF(X)'s and IB_X's at the two sites are the same except for the time intervals of message delay between S_i and S_j.

Further for each object at S_i, two more buffers are prepared by the LM to keep waiting requests. These are the <u>readers' buffer</u> (RB) and <u>writers' buffer</u> (WB). The RB (WB) which keeps the waiting read (write) requests for X is denoted by RB_X (WB_X), respectively. The write requests in each WB are sorted in the timestamp by the LM.

5. Correctness of the Algorithm

In this section, the correctness of the algorithm DMV is proved.

Theorem 5.1. The algorithm DMV preserves consistency.

Proof. Let **T**={T_1, ..., T_m} be the set of transactions which have been executed in the system so far by the algorithm DMV, excluding those aborted. Then, it is obvious that, as for the committed transactions, both of the database and the users observe the execution of these transactions which are equivalent to a serial execution of them consistent with the timestamp order. (Q.E.D.)

Theorem 5.2. No deadlock occurs by the algorithm DMV.

Proof. The following two cases are considered for a request to wait for its execution.

Case 1. A write request waiting for its a-lock to be granted.

A write request of T_i for an object X has to wait if X is already a-locked or c-locked by T_j, and TS(T_i)>TS(T_j).

Case 2. A read request waiting for an a-lock to be converted.

A read request for an object X by T_i has to wait if X is a-locked by T_j, and TS(T_i)>TS(T_j).

If we write the waits for graph with nodes of transactions and arcs representing "wait for" relation (see, e.g., [GRAY-78], [KING-73]), the arc from T_i to T_j is added to the graph only if TS(T_i)>TS(T_j) holds in both of above two cases. Thus, the waits for graph is acyclic and it was proved that no dead lock occurs if the wait for graph is acyclic (see, e.g., [GRAY-78], [KING-73]). (Q.E.D.)

Next, we shall show that no livelock occurs in the system.

Lemma 5.1. The value of MIN_i is monotonously increasing.

Proof. The value of MIN_i is the minimum timestamp of the active transactions at S_i. Assume that T^* is a transaction with timestamp MIN_i.

First, it is easy from the assumptions on the timestamps to see that MIN_i is not updated as long as T^* is active. Thus, MIN_i is updated when T^* becomes inactive or is aborted. At this point, as for the updation of MIN_i, the following two cases are considered:

Case 1. If there exist active transactions at S_i, the new value of MIN_i becomes the timestamp of a transaction T' with the minimum timestamp among those. By the assumption on T^*, $TS(T')>TS(T^*)$ holds.

Case 2. If no active transaction is at S_i, then C_i reads the local clock time and creates the timestamp (ts) which becomes the new value of MIN_i. In this case also, $ts>TS(T^*)$ holds. C_i repeats this process until a new active transaction arises at S_i. While the execution of such process, the values of MIN_i are monotonously increasing from the assumption on the mechanism of issuing timestamps.

Consequently, the lemma is proved. (Q.E.D.)

Lemma 5.2. The value of MINAC is monotonously increasing.

Proof. The value of MINAC is defined to be the minimum value of MIN_i, $(i=1, \ldots, N)$. Since the value of MIN_i is monotonously increasising from lemma 5.1, it is obvious that the values of MINAC are also monotonously increasing. (Q.E.D.)

Lemma 5.3. Read and write requests issued by a transaction whose timestamp is the value of MINAC is executed immediately.

Proof. Let T^* be the transaction such that $TS(T^*)$=MINAC, that is, T^* has the minimal timestamp among active transactions throughout the system. The following two cases in which requests are forced to wait for their execution are considered.

Case 1. A write request waiting its a-lock to be granted.

Assume that a write request of T^* for X has to wait because X is already a-locked or c-locked by a transaction T'. This means $TS(T^*)>TS(T')$. If T' is a-locking X, then T' is active, which contradicts that $TS(T^*)$ is MINAC. If T' is c-locking X, this means that $TS(T')=TS(X_{NN})>TS(X_N)$ and hence $TS(T^*)$=MINAC>$TS(X_N)$. Thus, the discard condition for X is satisfied, which contradicts that T' c-locking X. Consequently, a write request by T^* is not forced to wait for its execution.

Case 2. A read request waiting an a-lock to be converted.

Assume that a read request of T^* for X has to wait because X is a-locked by a transaction T'. This means T' is active and $TS(T^*)>TS(T')$ is satisfied, which contradicts that $TS(T^*)$ is MINAC. Thus, a read request by T^* is not forced to wait for its execution.

Therefore, the lemma is proved. (Q.E.D.)

Lemma 5.4. A transaction whose timestamp value is the MINAC is either committed or aborted within a finite time.

Proof. Let T^* be a transaction such that $TS(T^*)$=MINAC. From lemma 5.3, no read and write requests of T^* wait for their execution. That is, they are either granted immediately or rejected.

Therefore T^* is either committed or aborted within a finite time.

(Q.E.D.)

Lemma 5.5. The values of MINAC are updated monotonously increasingly within a finite time.

Proof. It is obvious from lemma 5.2 and lemma 5.4.

(Q.E.D.)

Theorem 5.3. Each transaction is either committed or aborted within a finite time after being initiated.

Proof. Suppose that a transaction T is now initiated. At this point, TS(T)$\geq$MINAC holds. If TS(T)=MINAC, then from lemma 5.4, T is either committed or aborted within a finite time. Thus, consider the case TS(T)>MINAC. Assume that T will be neither committed nor aborted within a finite time. This means T remains active infinitely. However, from lemma 5.5, TS(T) becomes the value of MINAC within a finite time, and then T is also committed or aborted within a finite time, which contradicts the assumption. (Q.E.D.)

Corollary 5.1. No livelock occurs by the algorithm DMV.

Proof. Theorem 5.3 proves this lemma. (Q.E.D.)

Now, we get to the goal.

Theorem 5.4. The algorithm DMV is correct.

Proof. It is straightforward from theorem 5.1 and theorem 5.3. (Q.E.D.)

6. Properties of Algorithm DMV

In this section, several important properties of the algorithm are discussed. First, the favourable property of the algorithm is shown by the following theorem.

Theorem 6.1. Any read request is granted.

Proof. By the discard condition, $X_O \leq MINAC$ is always satisfied for every object X. Since any read request has the timestamp larger than or equal to the current value of MINAC, there exists a version to be read by the request. (Q.E.D.)

From theorem 6.1, every read request is granted in our scheme by the discard condition with some waiting time for its execution if necessary. A transaction is backed up only if its write request is rejected. Thus, read requests have higher priority than write requests in our scheme.

Second, assume that two versions X_O and X_N of object X are in the system. Then if $TS(X_N) < MINAC$ is satisfied, it is meaningless to keep X_O because no read request for X reads the value of X_O. However, from reliability point of view, to keep X_O is useful if X_N is lost by an accident.

Finally, we discuss the locking mode. Our locking mode is so called a (r,a,c)-protocol [BAYE-80a], though r-lock is not employed in the algorithm. The "c" means that after the newest version is committed, it is accessible by other transactions. Now, let us consider an object X in state D3 and assume that many read requests are in the system. Then, the probability that one of these read requests has larger timestamp than the write requests waiting in the WB_X might be high. This means that many write requests would be rejected and hence many transactions would be backed up. In order to reduce abortion of transactions, these read requests should withold from their execution. The most extreme way to realize such mechanism is to prohibit the read requests from reading the newest version when the object is c-locked. Such locking mode is called the (r,a,x)-protocol [PEIN-83] and can be implemented for our model with a slight modification of the algorithm DMV. This modified version is rather conservative scheme and concurrency of the execution of the operations is reduced, while the algorithm DMV is rather optimistic. The interesting comparison of performace of these two locking modes is discussed in [PEIN-83].

7. Conclusion

A concurrency control scheme for a distributed database system is described. Different from the conventional distributed database models, in this paper, we try to distribute not copies of objects but versions of them. As the first step of such model, we consider the simplest model; namely, each object is located at different two sites, and they are the old version and the new one. An advantage of such scheme is increase of concurrency because of multiversion. In addition, updating of objects is simplified, since versions at different sites need not have the same value at all.

Taking actual environment into account, we employed timestamp mechanism for synchronization, which however reduces the concurrency of execution of operations due to the total order among transactions imposed in advance. Therefore, it will be a future work to develop the method which uses dynamic timestamp allocation or time intervals (see, e.g., [BAYE-81], [BAYE-82]), which is effective to improve concurrency. Abortion of write operations is useless since it backs up a transaction after a part of it has already executed. Thus, for our datadase model, to develop so called "cautious schedule" (see, e.g., [CASA-81], [KATO-84]), in which no transaction is backed up once its "begin operation" is accepted, is also important from the practical point of view. Finally, it is also an important future work to consider to extend the above model to a model where more than two versions of each object are distributed at different sites.

Acknowledgment

The authors wish to thank Professor T. Ibaraki of Toyohashi University of Technology, Associate Professor N. Katoh of Kobe University of Commerce, and Dr. S. Masuyama of Kyoto University for their useful discussions of this work. This work was supported in part by the Ministry of Education, Science and Culture of Japan under Scientific Research Grant-in-Aid.

References

[BAYE-80a] Bayer, R., Heller, H., and Reiser, A., "Parallelism and recovery in database systems", ACM Trans. Database Syst., Vol.5, No.2, pp.139-156 (June 1980).

[BAYE-80b] Bayer, R., Elhardt, K., Heller, H. and Reiser, A., "Distributed concurrency control in database systems", In Proc. 6th Int.. Conf. on VLDB, pp.275-284 (Oct. 1980).

[BAYE-81] Bayer, R., Elhardt, K., Heigert, J., and Reiser, A., "Dynamic timestamp allocation and its applications to the BEHR-method", Tech. Rep., Technical University Munich (July 1981).

[BAYE-82] Bayer, R., Elhardt, K., Heigert, J., and Reiser, A., "Dynamic time-stamp allocation for transactions in database systems", In Distributed Data Bases, H.-J. Schneider (ed.), North-Holland, pp.9-20 (1982)

[BERN-81] Bernstein, P.A. and Goodman, N., "Concurrency control in distributed database systems", *ACM Comput. Surv.*, Vol.13, No.2, pp.185-222 (June 1981).

[BERN-83] Bernstein, P.A. and Goodman, N., "Multiversion concurrency control - Theory and algorithms", *ACM Trans. Database Syst.*, Vol.8, No.4, pp.465-483 (Dec. 1983).

[BRZO-84] Brzozowski, J.A. and Muro, S., "On serializability", Tech. Rep. #840012, Dept. of Applied Math. and Physics, Faculty of Engineering, Kyoto University (July 1984).

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

[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", *Comm. ACM*, Vol.19, No.11, pp.624-633 (Nov. 1976).

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

[IBAR-83] Ibaraki, T., Kameda, T., and Minoura, T., "Disjoint-interval topological sort: a useful concept in serializability theory", In *Proc. 9th Int. Conf. on VLDB*, pp.89-91 (Oct./Nov. 1983).

[KATO-84] Katoh, N., Ibaraki, T., and Kameda, T., "Cautious transaction schedulers with admission control", TR 84-2, Dept. of Computer Science, Simon Fraser University (Feb. 1984).

[KING-73] King, P.F. and Collmeyer, A.J., "Database sharing - an efficient mechanism for supporting concurrent processes", In *Proc. National Computer Conference*, pp.271-275 (1973).

[LAMP-78] Lamport, L., "Time, clocks, and the ordering of events in a distributed systems", *Comm. ACM*, Vol.21, No.7, pp.558-565 (July 1978).

[MIZU-84] Mizutani, T., "Multiversion concurrency control scheme for a distributed database system", Master Thesis, Dept. of Applied Math. and Physics, Faculty of Engineering, Kyoto University (Feb. 1984).

[MURO-84] Muro, S., Kameda, T., and Minoura, T., "Multi-version concurrency control scheme for a database system", To appear in *J. Comput. Syst. Sci.* (1984).

[PAPA-79] Papadimitriou, C.H., "The serializability of concurrent database updates", *J. ACM*, Vol.26, No.4, pp.631-653 (Oct. 1979).

[PAPA-84] Papadimitriou, C.H., and Kanellakis, P.C., "On concurrency control by multiple versions", *ACM Trans. Database Syst.*, Vol.9, No.1, pp.89-99 (March 1984).

[PEIN-83] Peinl, P. and Reuter, A., "Empirical comparison of database concurrency control schemes", In *Proc. 9th Int. Conf. on VLDB*, pp.97-108 (Oct./Nov. 1983).

[REED-78] Reed, D.P., "Naming and synchronization in a decentralized computer system", Tech. Rep. MIT/LCS/TR-205, Dept. of EECS, MIT (Sept. 1978).

[ROSE-84] Rosenkrantz, D.J., Stearns, R.E., and Lewis II, P.M., "Consistency and serializability in concurrent database systems",*SIAM J. Comput.*, Vol.13, No.3, pp.508-530 (Aug. 1984).

[RUSS-80] Russel, D.L., "State restoration in systems of communicating processes", *IEEE Trans. Softw. Eng.*, Vol.SE-6, No.2, pp.183-194 (March 1980).

[STEA-81] Stearns, R.E., and Rosenkrantz, D.J., "Distributed database concur-rency using before-values", In *Proc. ACM-SIGMOD*, pp.74-83 (April/ May 1981).

[SUGI-84] Sugihara, T., Kikuno, T., and Yoshida, N., "Deadlock detection and recovery in distributed database systems", (in Japanese), *The transactions of the institute of electronics and communication engineers of Japan*, Vol.J67-D, No.1, pp.1-9 (Jan. 1984).

[THOM-79] Thomas, R.H., "A solution to the concurrency control problem for multiple copy databases", In *digest of papers IEEE COMPCON spring*, pp.56-62 (Feb./March 1978).

[ULLM-80] Ullman, J.D., "Principles of database systems", *Computer Science Press* (1980).

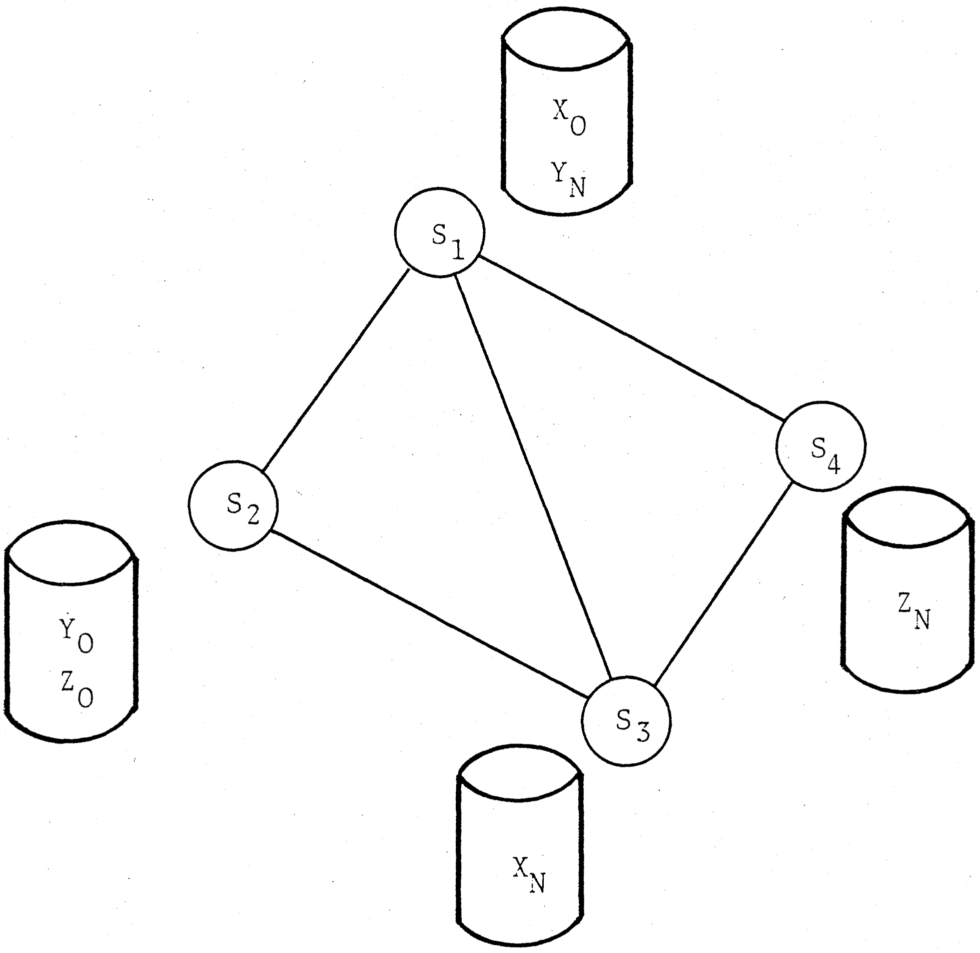

Fig.1 - An example of distribution of versions in a system.

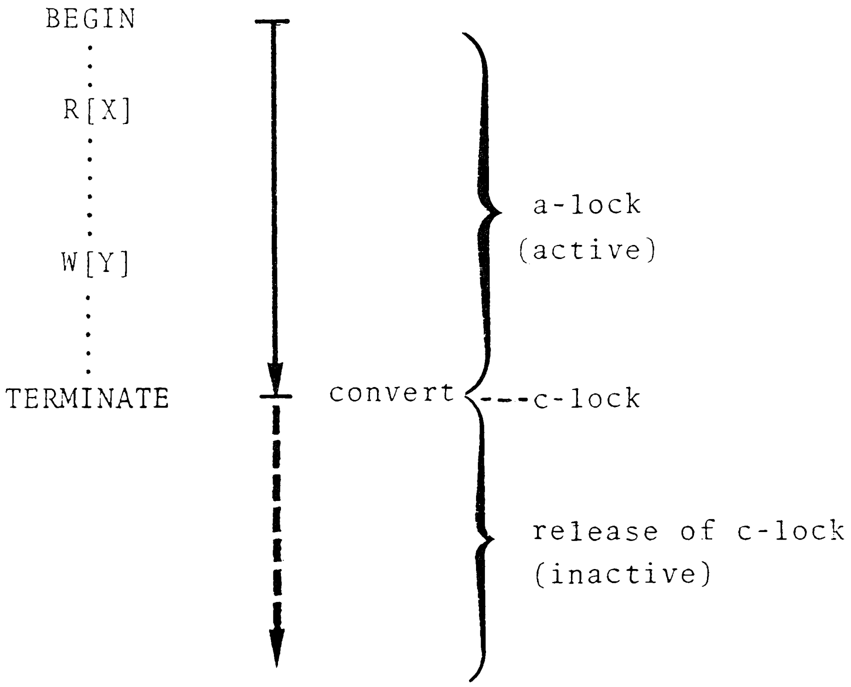

Fig.2 - The execution of a transaction.

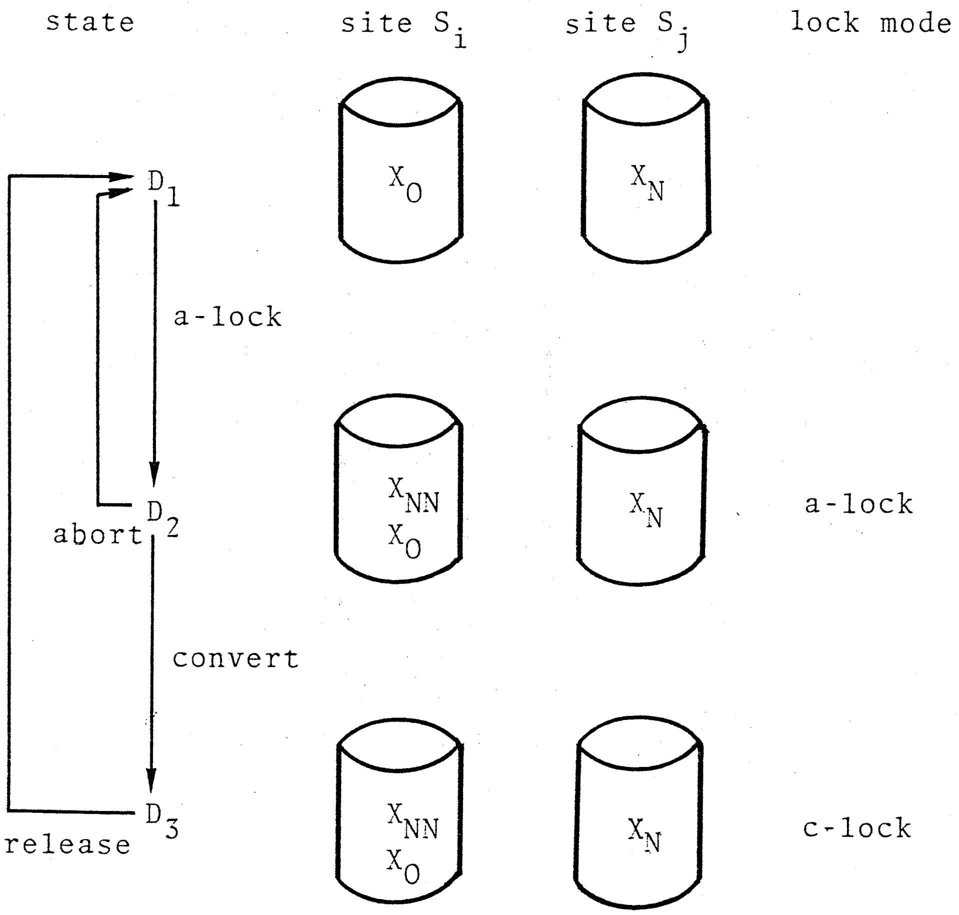

Fig.3 - The state transition of an object X.

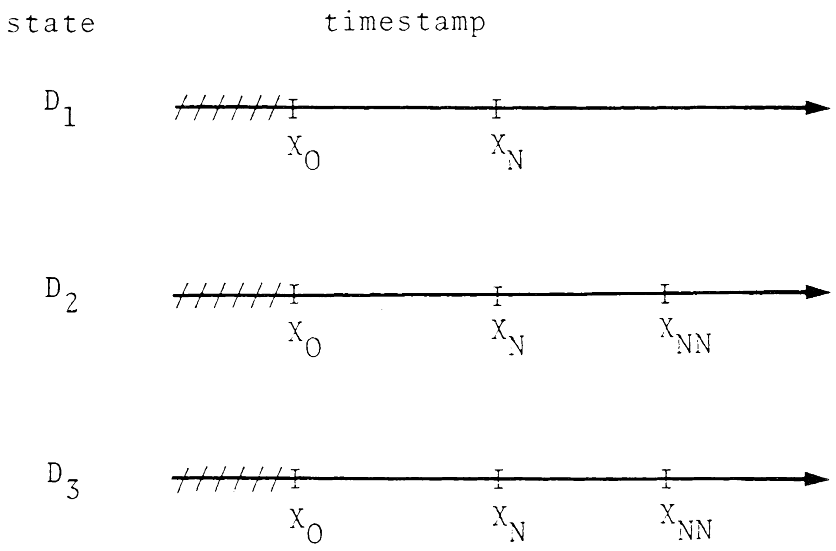

Fig.4 - A snapshot of the state of an object X concerning timestamp.

name	value	writer	readers	max-reader	relative version number

Fig.5 - The configuration of a version.