Global concurrency control mechanisms for a local network consisting of systems without concurrency control capability

by YAHIKO KAMBAYASHI

Kyushu University*
Fukuoka, Japan

and

SEI-ICHI KONDOH

Mitsubishi Electric Co.
Kamakura, Japan

ABSTRACT

A powerful and expandable system can be economically realized by a local computer network consisting of various kinds of microprocessor-based systems. The following three problems must be solved to organize a distributed processing system using nonidentical elements: (1) communication, (2) query conversion, and (3) global concurrency control. Except in the case when all transactions are read-only ones, (3) must be handled. Since each system in a network does not usually have concurrency control capability or may not use the identical mechanism, it is necessary to develop a global concurrency control mechanism for a local network consisting of systems without such capability. In this paper two such mechanisms are presented. By assigning ordered numbers to the component systems, a consistent and deadlock-free global mechanism is realized for a semijoin-based query procedure. To improve efficiency, a mechanism permitting dynamic modification capability of ordering is also presented.

*This paper was written when the authors were at Kyoto University.
INTRODUCTION

The top-down and the bottom-up approaches are those that can be used to organize a distributed system. Through the former approach, which takes a global view of the whole system, consistent and efficient systems can be easily designed. This paper, however, will discuss the latter approach, since it is a practical solution to the problem of constructing a distributed system using already existing systems, such as work stations with database capability, database machines, and picture file systems using laser discs. For this approach the following problems must be solved: (1) communication procedures among systems, (2) conversion of user requests, and (3) global concurrency control. Except for the case when all transactions are read-only ones, Problem 3 must be solved. Since Problems 1 and 2 are handled by various authors, this paper will focus on Problem 3.

As described below (see Figure 1(a)), Problem 3 must be considered even when there exists no global write transaction. That is, considering only query processing procedures is not enough to handle global read-only transactions when local write transactions at each site are permitted. Although this problem is very important when constructing a network using various different subsystems, the authors believe that it has not been discussed before. To simplify the problem, we will use the following three restrictions, which are considered to be reasonable:

1. To avoid Problems 1 and 2, we assume that the component systems realize relational databases with an identical query language.
2. For a network we only consider an Ether-type local network with broadcasting capability.
3. We decompose a global transaction into a global read-only transaction and local read-write transactions so that global read-write transactions can be avoided. Since handling of such a global read-write transaction makes discussion complicated, it is excluded in order to present basic ideas.

For the global concurrency control problem, the following three cases must be considered:

1. There may be a system that does not have any concurrency control mechanisms. All transactions are proposed serially at this system, and local locking mechanisms are not available to global control mechanisms.
2. Even if a system has a concurrency control mechanism, it may not be usable for global control. That is, there may be a system with an independent concurrency control mechanism in order to improve efficiency at its own site, which is not suitable for distributed control.
3. Even if all the systems have global concurrency control mechanisms, they may not be identical. For example, some systems use time-stamp-based mechanisms, whereas other systems employ two-phase lock mechanisms. We cannot combine these different global concurrency control mechanisms.

Since Case 1 is the most restrictive, this paper will discuss that case. Since the locking mechanism is not available for global concurrency control, a query modification approach is used to realize such control.

Figure 1(a) shows that we need a global concurrency control mechanism when all global transactions are read-only. We will consider the following four transactions where $t_1$ and $t_2$ are global read-only ones and $t_3$ and $t_4$ are local write transactions:

- $t_1$: After reading value A at site 1, read value D at site 2.
- $t_2$: After reading value C at site 2, read value B at site 1.
- $t_3$: Modify values A and B at site 1.
- $t_4$: Modify values C and D at site 2.

The order of the transaction processing at sites 1 and 2, by the schedule shown in Figure 1(a), is as follows:

- site 1: $t_1$ $\rightarrow$ $t_2$ $\rightarrow$ $t_3$
- site 2: $t_2$ $\rightarrow$ $t_4$ $\rightarrow$ $t_1$

Since these two orderings are not compatible, we have to restart either $t_1$ or $t_2$. This example shows that we need a global concurrency control mechanism even if all transactions are read-only.

In order to avoid such a problem, the order of transaction processing at each site should be controlled by a global concurrency control mechanism. We will present a global concurrency control mechanism that uses ordering numbers assigned to the sites. A semijoin-based query-processing procedure is combined with the mechanism. Another mechanism is also presented, which improves efficiency by modifying the site ordering numbers adaptively.
BASIC CONCEPTS

Concurrency Control

For efficient processing, it is important to execute many transactions concurrently. In this case a semantically correct schedule must be generated. Here a schedule consists of a sequence of read and write operations (see Figure 1(a)). Generally we shall assume that the schedule is consistent if and only if its effect is equivalent to that obtained by executing the same transactions serially in some order, called serializable.1 We say that two schedules are equivalent if and only if the value that one transaction reads was written by the same transaction in both schedules. For example, in Figure 1(a), t2 reads the value that t3 wrote at site 1, so t2 must be before t3 in an equivalent serial schedule. The graph in Figure 1(b) shows this kind of precedence relationship among transactions. Since it has a cycle, there is no equivalent serial schedule; that is, this schedule is not serializable. To guarantee serializability, many methods have been introduced. In centralized database systems, concurrency control mechanisms are not essential, since transactions can be executed serially. In distributed database systems, however, since transactions are executed in parallel at several sites, a global concurrency control mechanism is necessary even if each site has a local concurrency control mechanism. When many transactions are executed concurrently, deadlock may occur; so a deadlock-free mechanism is also required.

One of the methods used to ensure that schedules are serializable and deadlock-free is the tree protocol.2 If each transaction obeys the tree protocol, no global scheduler is required. The relationship among data is assumed to be represented by a tree, which is true for hierarchical database systems.

The basic operations to be considered are LOCK and UNLOCK. Only one transaction is permitted to lock a datum at a time. We use L(A) and U(A) to represent LOCK A and UNLOCK A, respectively. A tree protocol is satisfied by a transaction with respect to T, a tree whose nodes corresponds to data if

1. Any datum can be locked for the first time.
2. A datum can be locked if its parent is currently locked.
3. Any datum can be unlocked at any time.
4. No datum is ever locked twice by one transaction.
5. Transactions requiring access to data at different levels of the tree structure must lock each record connecting the different levels.

Example 1: We will consider the four transactions used in Figure 1. We assume that the tree showing the relationships among data is shown as Figure 2. In order to obey the tree protocol, transactions are modified as follows:

\[
t_1: L(A)L(C)U(A)L(D)U(C)U(D)
\]

\[
t_2: L(C)L(B)U(C)U(B)
\]

\[
t_3: L(A)L(C)U(A)L(B)U(C)U(B)
\]

\[
t_4: L(C)L(D)U(C)U(D)
\]

Although t1 requires A and D only, it has to lock C because of the requirement 2 of the tree protocol. Since the first lock of a transaction is not restricted, t2 and t3 start by locking C.

Query Processing

Let R be a relation on a set \(\{A_1, \ldots, A_n\}\) of attributes, where the set is denoted by \(R\), the relation schema of \(R\). Let \(u\) be a tuple of a relation and \(u[X]\) be the part of \(u\) corresponding to the attribute set \(X\). In this paper the following notations of relational algebra will be used:

\[
\text{Projection: } R[X] = \{u[X] \mid u \in R\}
\]

\[
\text{Natural equijoin: } R_1 \bowtie R_2 = \{u \mid u \in R_1, u[R_1] \in R_2, u[R_1] \in R_1, R = R_1 \bowtie R_2\}
\]

A query graph \(G_q = (V, E, L)\) corresponding to a natural join query \(q\) is a labeled undirected graph. \(V\) is a set of vertices, where \(v_i\) in \(V\) corresponds to relation \(R_i\), referred to in \(q\). Two vertices \(v_i\) and \(v_j\) corresponding to \(R_i\) and \(R_j\) are connected by an edge if and only if there is \(R_i \bowtie R_j\). The label of the edge is a subset of \(R_i \bowtie R_j\), \(E\) is the set of edges, and \(L\) is the set of labels for \(E\).

A query is called a tree query if there exists a query graph that corresponds to it and it is circuit-free; otherwise it is cyclic.

A semijoin of \(R_i\) by \(R_j\) is denoted by \(R_i \bowtie< R_j\) and defined as

\[
R_i \bowtie< R_j = (R_i \bowtie R_j)[R_j]
\]

In distributed database systems semijoins are used in order to reduce the cost of communications. For tree queries there exists an efficient procedure to calculate partial results for all relations using semijoins only. Here a partial result for \(R_i\) is the result of the join projected on \(R_i\). Since conversion methods exist which can transform cyclic queries into tree queries,4,5 we consider tree queries only in this paper.

Although there may be more than one relation at each site, for simplicity we assume that each site \(S_i\) contains exactly one relation \(R_i\), which is obtained by preprocessing all relations at site \(S_i\) involved in the query. This assumption is commonly used, and the scheme shown here may be easily extended to handle more general cases.

A general semijoin-based tree query-processing procedure is as follows (because of space limitations, we have simplified the description):
Procedure 1: Query-processing procedure for a tree query using semijoins.

1. In the tree graph representing the given query, select an arbitrary relation as a root of the tree.
2. Phase 1: Starting from the leaf relations, perform semijoins by sending values of join attributes.
3. At the root relation, a partial result is obtained.
4. Phase 2: Starting from the root relation, perform semijoins by sending values of join attributes. At each site partial results are then obtained.

Example 2: Let us consider the tree query in Figure 3. The attributes of the relations are as follows:

\[ R_1(AD) \quad R_2(ABCE) \quad R_3(BF) \quad R_4(CG) \]

We assume that each \( R_i \) is stored at site \( S_i \) (\( i = 1,2,3,4 \)). The following \( R \) is required as the result:

\[ R = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \]

Partial results for these relations are as follows:

\[ R[AD] \quad R[ABCE] \quad R[BF] \quad R[CG] \]

If only partial results are required, the semijoin-based algorithm is sufficient. However, if \( R \) is required at some site, the algorithm can be used as a preprocess.

1. We can select any relation as a root. Let \( R_1 \) be the root.
2. Phase 1:
   (2-1) Send \( B \) values of \( R_1 \) from site \( S_1 \) to site \( S_2 \). Perform a semijoin with \( R_2 \).
   (2-2) Send \( C \) values of \( R_1 \) from site \( S_1 \) to site \( S_3 \). Perform a semijoin with the result of (2-1).
   (2-3) Send \( A \) values of the result of the above two operations from site \( S_2 \) to site \( S_1 \). Perform a semijoin with \( R_1 \).
3. At site \( S_1 \) the partial result \( R[AD] \) is obtained.
4. Phase 2:
   (4-1) Send \( A \) values of the above result to site \( S_2 \). Perform a semijoin and the partial result \( R[ABCE] \) is obtained at site \( S_2 \).
   (4-2) Send \( B \) (and \( C \)) values of \( R[ABCE] \) to site \( S_1 \) (and site \( S_4 \), respectively). By performing a semijoin the partial result \( R[BF] \) (and \( R[CG] \)) can be obtained at site \( S_1 \) (and site \( S_4 \), respectively).

THE PROBLEM

Although an overview of some of the problems involved with global concurrency control, as well as the assumptions made, were discussed in the introduction, we will give specific details here.

Consider the case when systems without concurrency control mechanisms are connected by an Ether-type bus line. This network satisfies the following properties: (1) a message can be broadcast to all the sites, and (2) it is not possible to transmit messages simultaneously from more than one site.

All global transactions are assumed to be read-only. Modification of relations is assumed to be realized by local transactions. This is similar to a relational database system that realizes views. Usually, however, modification operations are permitted to be applied to the base relations only (read-only views), because the general view update problem is known to be very difficult.

Since semijoin-based query-processing procedures are very efficient, we will use them in this paper. We have to modify the procedure, however, because of the following problem.

If the data are modified between Phase 1 and Phase 2 in Procedure 1, we may not get the correct result. If we organize a distributed database system by the top-down approach, we usually use a locking mechanism to prevent such a modification. Since the assumption that the subtransaction at Phase 1 and one at Phase 2 are considered to be different at the processing site, data may be modified before the second subtransaction. To handle the problem caused by such a local write transaction, one simple method is to store the values at Phase 1 that will be used at Phase 2. This approach, however, may require many duplicated data, and there still exists a global consistency problem (Introduction); so we will discuss methods to prevent this problem in following sections.

We will consider query-processing procedures together with concurrency control mechanisms. Usually read-only transactions are called queries. In the following sections we use the term query instead of transaction when a transaction performs only read requests.

A QUERY-PROCESSING PROCEDURE AND A BASIC GLOBAL CONCURRENCY CONTROL MECHANISM

As shown in the previous section, it is necessary to modify the semijoin-based query-processing procedure when local write transactions are permitted. In this section we will present a query-processing procedure having the following properties.

1. Instead of visiting the same site twice at Phases 1 and 2, it requires that each site be visited only once.
2. Relations in a query tree can be processed in an arbitrary order.

We need the first property because of the problem pointed out in the previous section. The second property is used to combine the query-processing procedure with a tree-protocol-based global concurrency control mechanism.
First we modify the basic semijoin-based procedure to satisfy Property 1 above.

**Example 3:** Let us consider the same query as Example 2. We assume that the target relation \( R[ABG] \) is required at site \( S_1 \). By sending values contained in \( ABG \) together with the join attributes, the result can be obtained at site \( S_1 \) by performing Phase 1 only.

1. Let \( R_1 \) be the root.
2. (2-1) Same as Example 2.
   (2-2) Send \( R_4(CG) \) to site \( S_2 \), since \( C \) is the join attribute and \( G \) is contained in the target. Perform a join.
   (2-3) Send combined values of \( ABG \) to site \( S_1 \) and perform a join.
3. At site \( S_1 \), \( R[ABG] \) is obtained.

We assume that projection \( R[X] \) of the join of all relations in the query is required at one site. In such a case we only need Phase 1 of Procedure 1 by transmitting attributes in \( X \) together with join attributes.

A tree query is usually processed from leaf sites by Procedure 1; but by using the broadcast capability of Ether-type networks, we can change the order of processing.

**Procedure 2:** Query-processing procedure using the broadcast capability.

1. Let \( T \) be the tree representing the given query.
2. Select one arbitrary relation \( R_i \) in \( T \). Let \( X \) be the attribute set of the target relation (i.e., \( R[X] \) is required at site \( S_t \), where \( R \) is the join of all relations involved in the query). Let \( Y \) be the union of attributes satisfying
   \[
   Y = R_i \cap (X \cup R_j)
   \]
   where \( \cup R_j \) denotes the union of all the join attributes of \( R_j \). Broadcast \( R_i[Y] \) to all sites.
3. Let \( R_1, \ldots, R_m \) be all relations satisfying \( R_i \cap R_j \neq \emptyset \). Let \( R_k \) be one of the relations.
   (3-1) Except \( R_k \), perform the following semijoin at the site of \( R_j \) (\( j = j_1, \ldots, j_m \)).
   \[
   R_k \bowtie \sigma R_j[R_i \cap R_j]
   \]
   (3-2) At the site of \( R_k \) perform the following join:
   \[
   R_k \bowtie \sigma R_i[Y]
   \]
   Note that attribute set of \( R_k \) may change if \( R_i \) contains attributes in \( X \) that were not originally contained in \( R_k \).
4. Let \( T' \) be the new tree obtained from \( T \) by eliminating \( R_k \). \( T' \) can be obtained by the following steps:
   (4-1) Remove all edges connecting between \( R_i \) and \( R_j \) (\( j = j_1, \ldots, j_m \)) directly. Remove \( R_k \).
   (4-2) Connect \( R_i \) and \( R_j \)'s (\( j = j_1, \ldots, j_m, j \neq k \)) directly.

The conversion of (4-1) and (4-2) is shown in Figure 4(a) and (b). Let \( T' \) be the new \( T \) and goto step (2).

5. Repeat the above process until only the relation at the target site remains. At that time the result \( R[X] \) is obtained at the site.

**Theorem 1:** Procedure 2 is correct.

**Proof:** We only need to prove that the transformation shown in Figure 4(a) and (b) is correct. It is easily shown by the following equation.

\[
R_1 \bowtie \sigma R_2 \bowtie \cdots \bowtie \sigma R_m = (R_1 \bowtie \sigma R_2) \bowtie \cdots \bowtie \sigma (R_m \bowtie \sigma R_k)
\]

QED

One possible problem of Procedure 2 is to find a method to determine \( R_k \) in Step 3.

**Procedure 3:** Select of \( R_k \) at Step 3 of Procedure 2.

1. Let \( S_0 \) be the site where it is required to obtain the target relation.
2. Among \( R_j \)'s select \( R_k \) which is close to \( R_0 \). Here distance on the tree is determined by the number of edges between the two nodes.

**Example 4:** Let us consider the tree query shown in Figure 3. We assume that the target relation \( R[ABG] \) is required at site \( S_1 \), \( X = ABG \).

1. Let \( R_2 \rightarrow R_1 \rightarrow R_4 \rightarrow R_3 \) be the linear order of the relations; we assume that relations are processed in this order.
2. \( R_2 \):
   (2-1) Select \( R_1 \) as the relation to apply to the join.
   (2-2) Broadcast \( R_2[ABC] \) (\( A, B, \) and \( C \) are join attributes) and perform a join at \( S_1 \) and semijoins at \( S_3 \) and \( S_4 \). The resulting relation at each site is as follows:
   \[
   R_1(ABCD) \quad R_2(BF) \quad R_2(CG)
   \]
   (2-3) By eliminating \( R_2 \), a new query graph, shown in Figure 5, is obtained.
   (2-4) Process \( R_3 \) and \( R_4 \) by the conventional tree-query-processing procedure.
3. At site \( S_1 \), \( R[ABG] \) is obtained.
Since for any query we can use the same order to process relations, the following concurrency control mechanism can be used:

**Procedure 4**: Query-processing procedure for a tree query preserving global consistency.

1. There is a fixed ordering of the sites. Let $S_1, S_2, \ldots, S_n$ be the sequence of the sites in this order. Since we assume that each site does not process queries concurrently, at any moment each site processes at most one query.

2. For each query we use the ordering $(S_1, S_2, \ldots, S_n)$ to select sites by Procedure 2. We must consider the following two cases in order to apply Procedure 2.
   - (2-1) After processing $R_i$, we must proceed to $R_{i+1}$, but there are cases when $R_{i+1}$ is not contained in the query. In such cases we need put dummy processing of $R_{i+1}$ in the query.
   - (2-2) The target site $R_t$ may not be $R_n$, which is the last relation to be processed. In such a case we apply Procedure 2 as if $S_n$ is the target site. After obtaining the result at the site it is transmitted to the target site.

3. After processing $R_i$, if $S_{i+1}$ is occupied by another query, wait until it completes. When $R_i$ is assigned to process query $q_i$, it starts to perform joins received from $R_j$ ($j < i$) for $q_i$ (see Example 4).

The pipeline processing is achieved by (1) serial processing at each site and (2) the serial processing property of the communication bus line.

**Theorem 2**: Procedure 4 is correct, and it ensures serializability and deadlock freedom.

**Proof**: In the method shown in Procedure 4, a mechanism similar to a special case of the tree protocol is used. Data items are replaced by sites, and the chain showing the ordering $(S_1, \ldots, S_n)$ is a special tree structure. Thus, serializability and deadlock freedom result from the fact that the tree protocol satisfies these conditions.

QED

**Example 5**: Let us consider the tree query shown in Figure 6(a). Here attributes of relations are as follows.

$R_1(\text{AD})$  $R_2(\text{ABCE})$  $R_3(\text{BF})$  $R_4(\text{CG})$

We assume that each $R_i$ is stored at site $S_i$ ($i = 1, 2, 3, 5$) and that the joins of all relations is $R$. $R$ is required to be calculated at site $S_2$. We give a fixed linear order as follows:

$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$

$S_1$: Send $R_1$ to $S_2$. The resulting query graph is shown in Figure 6(b).

$S_2$: Perform a join. Select $R_3$ as $R_k$ and broadcast $R_1 \bowtie\bowtie R_2$. At site $S_3$, $R_3 \bowtie\bowtie R_2$ is stored. The computation starts when $S_3$ becomes the site to process the query.

$S_3$: Perform a semijoin and $R_3 \bowtie\bowtie (R_1 \bowtie\bowtie R_2)$ is obtained. Broadcast $R_3 \bowtie\bowtie (R_1 \bowtie\bowtie R_2)$.

$S_4$: Since $R_4$ is not contained in the query, it is used to synchronize with other queries. No computation is made at $S_4$.

$S_5$: Perform a join ($R_1 \bowtie\bowtie R_2$ from $S_2$, $R_3 \bowtie\bowtie (R_1 \bowtie\bowtie R_2)$ from $S_3$, and $R_4$ at $S_5$) and $R$ is obtained. Send its result to the $S_2$.

**A GLOBAL CONCURRENCE CONTROL MECHANISM USING ADAPTIVE ORDERING OF SITES**

In the previous section we discussed a basic global concurrency control mechanism. Since every query has to visit all sites in a fixed order between the first site and the last site required, it has the following disadvantages: (1) for each query the optimal ordering is usually different, so the cost for processing may become high; and (2) there are queries that need to visit only a few sites. The base mechanism may require that several sites be visited which are not used. This produces unnecessary overhead.

In this section we will develop a mechanism that changes
ordering adaptively according to the query set. As the network has broadcasting capability, each site can know the status of processing at other sites as well as queries in the queue. By the new mechanism, the ordering of the sites is modified according to the queries in the queuing list.

We will define a graph showing the order of the sites.

**Definition 1:** An order graph \( G_0(V,E) \) is a directed graph. \( V \) is a set of vertices, where \( v_i \) in \( V \) corresponds to site \( S_i \). \( E \) is a set of all directed edges. If there exists an edge \( e_{ij} \) from \( v_i \) to \( v_j \), site \( S_i \) precedes \( S_j \) in order. Let \( V_0 \) be vertices in \( V \) that have an incoming edge and/or an outgoing edge.

For the first query for the system we can determine an arbitrary order of sites. For the second query, sites that are not used by the first query can be processed in an arbitrary order. The order determined by the queries currently processed is shown by the order graph in Definition 1. When a new query is added or a query is completed, we can change the graph to improve the efficiency as compared to the fixed-order approach. We assume that the order graph is kept by all the sites.

**Procedure 5:** Procedure for the order graph modification.

Let \( V_0 \) be a set of vertices corresponding to the sites currently involved in the query processing. A subset \( V_m \) of \( V_0 \) determined by Case 2 is called a set of vertices whose orders are modifiable. Initially \( V_0 = \emptyset \) and \( V_m = \emptyset \).

**Case 1:** When a new query \( Q \) is added.

Let \( S \) be the set of sites used by \( Q \). We can determine the ordering of sites as follows:

1. For sites in \( S \cap (V_0 - V_m) \) the order does not conflict with the current order graph.
2. For the sites \( S \cap V_m \), the following graph modification process can be applied:
   1. Let a vertex in \( S \cap V_m \) be \( v_i \). We assume that there are edges \( e_{ai} \) and \( e_{ib} \). By the condition implied when generating a vertex in \( V_m \), each vertex in \( V_m \) has an outgoing edge. Add edge \( e_{ib} \).
   2. Remove \( e_{ai} \) and \( e_{ib} \). If \( v_i \) does not have incoming edges, Step 1-2-1 is not necessary.
   3. The position of \( v_i \) is arbitrary if the new position is the successor of the old \( v_i \).

**Case 2:** When a query \( Q \) terminates, sites used by \( Q \) only may be eliminated from the graph. There are the following two cases.

1. The site used by \( Q \) only has an outgoing edge. We cannot eliminate the vertex corresponding to the site, but the position of the vertex can be moved. We put the vertex to \( V_m \).

(2-2) The site used by \( Q \) only does not have an outgoing edge. In such a case we can eliminate the vertex corresponding to the site from \( V_m \). This process is applied recursively until no further elimination is possible.

**Example 4:** We assume that the following queries \( Q_1 \) and \( Q_2 \) are currently processed in the system:

\( Q_1: \) It uses \( S_1 \) and \( S_2 \) in the order \( S_1 \rightarrow S_2 \)
\( Q_2: \) It uses \( S_3, S_4, \) and \( S_5 \) in the order \( S_3 \rightarrow S_4 \rightarrow S_5 \)

We assume that \( Q_3 \) which uses \( S_1, S_2, S_4, \) and \( S_5 \) is added to the system.

\[ V_0 = \{S_1, S_2, S_3, S_4, S_5\} \]
\[ V_m = \emptyset \]
\[ S = \{S_1, S_2, S_3, S_4\} \]

For \( S \cap V_0 = \{S_1, S_2, S_3\} \), we must follow the orders determined by the queries \( Q_1 \) and \( Q_2 \), that is \( S_1 \rightarrow S_2 \). The order for \( Q_3 \) must not conflict with \( S_1 \rightarrow S_2 \). Let the order for \( Q_3 \) be

\[ S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4 \]

By merging these orders we get the following order:

\[ S_3 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4 \]

Now we assume that \( Q_2 \) terminates. Sites used by \( Q_2 \) only are \( S_3 \) and \( S_5 \). Since \( S_3 \) does not have outgoing edges, it can be eliminated from \( V_0 \). Since \( S_5 \) has an outgoing edge, \( V_m = \{S_3\} \).

An outline of the proof of the correctness of Procedure 5 is as follows. For any currently executing queries the visitation order is the same, so the process is the same as that in Procedure 4. The problem is caused by queries that have already terminated when query \( Q \) is added. We assume that the last site of \( Q \) is \( S_1 \). If such a query terminates at \( S_1 \)'s descendant, it is obvious that it is before \( Q \) in the equivalent serial schedule, so there is no contradiction. If it terminates at \( S_1 \)'s ancestor, the sites that it used cannot become the descendant of sites used by \( Q \), so no contradiction occurred.

**SUMMARY**

In this paper we have shown global concurrency control mechanisms for a local network consisting of systems that do not have concurrency control capability. Because of this assumption we do not use a locking mechanism at each site. The whole query is decomposed into subqueries at the site where the query is produced. Since the data flow control can be expressed in a query, the whole mechanism can be realized by a so-called query modification approach. The major reasons why we do not need locking or timestamp mechanisms are that (1) each site queries are serially processed and (2) by observing the data transmitted on the bus line, the status of the processing can be determined.
ACKNOWLEDGMENT

The authors are grateful to Professor Shuzo Yajima and members of Yajima Laboratory for helpful discussions.

REFERENCES
