Concurrency control principles in distributed and centralized databases

Georges Gardarin, M. Melkanoff

INRIA
ENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports de Recherche

N° 113

CONCURRENCY CONTROL PRINCIPLES IN DITRIBUTED AND CENTRALIZED DATABASES

Georges GARDARIN
Michel MELKANOFF

Janvier 1982

CONCURRENCY CONTROL PRINCIPLES IN DISTRIBUTED AND CENTRALIZED DATABASES

by

Georges Gardarin

INRIA - SIRIUS and University of Paris VI

and

Michel A. Melkanoff

University of California, Los Angeles

July 1981

CONCURRENCY CONTROL PRINCIPLES IN DISTRIBUTED AND CENTRALIZED DATABASES

Georges GARDARIN
INRIA, Projet SABRE
BP 105
78150 LE CHESNAY

Michel MELKANOFF
Computer Science Dept.
UCLA
CA90024, LOS ANGELES

Abstract

The purpose of this paper is to discuss one of the most controversial problem arising in distributed databases, namely concurrency control, and to present in a unified way the various control techniques which have been proposed to resolve this problem. Also, we introduce some basic concurrency control principles and we derive from them correctness proofs of the various presented algorithms. The presented methods are classified in three typical groups : data granule locking, prefixed ordering of transaction accesses, and other ordering techniques. Finally, the qualities of the various approaches are compared.

Résumé

Le but de cette communication est de discuter un des problèmes les plus controversés soulevé par les bases de données réparties et centralisées : le controle de concurrence. Une vue unifiée des diverses techniques de controle est presentée. Aussi, nous introduisons des principes de base et nous dérivons de ces principes des preuves de corrections des algorithmes présentés. Ces algorithmes sont classés en trois groupes généraux basés sur le verrouillage de granules, l'ordonnancement à priori des accés des transactions et les autres types d'ordonnancement. Les qualités des diverses approches sont comparées.

CONTENTS

1 INTRODUCTION 1

2 DEFINITIONS AND PROBLEM STATEMENTS 5

2.1 Data Integrity 5

2.2 Concurrency Problems 7

2.2.1 Lost Operations 7

2.2.2 Inconsistencies 8

3 CHARACTERISTICS OF CONFLICT-FREE EXECUTIONS 10

3.1 Basic Concepts 10

3.2 Scheduling of Distributed Transactions 12

3.3 Serializable Schedules 15

3.4 Properties of Operations 16

3.5 Characteristics of Serializable Schedules 17

3.6 Precedence Graphs 19

4 ALGORITHMS BASED ON THE INITIAL ORDERING OF TRANSACTIONS 22

4.1 Transactions Timestamps 22

4.2 Granule Timestamps 24

4.3 Total Ordering Algorithms 25

4.4 Partial Ordering Algorithms [BERN80] 27

4.5 Multiversion Partial Ordering Algorithms [BERN80] 28

5 TWO-PHASE LOCKING ALGORITHMS 30

5.1 Basic Principles 30

5.2 Locking Protocols 31

5.3 Locking Algorithms 33

5.4 Two-phase Restriction [ESWA76] 35

5.5 Locking in a Distributed System 38

5.6 The Deadlock Problem 39

5.6.1 Representations of deadlock 40

5.6.2 Detection of deadlock in a centralized control system 44

5.6.3 Detection of deadlock in a distributed control system 46

5.6.3.1 Centralized execution of the detection algorithm 47

5.6.3.2 Construction of the waiting graph 47

5.6.3.3 Inquiry initiation 48

5.7 Other Problems Due to Locking 49

6 OTHER CONCURRENCY CONTROL ALGORITHMS 52

6.1 Ordering by Validation of Reads Before Writes 52

6.2 Detection of Circuits in Precedence Graphs 54

7 CONCLUSION 55

1. INTRODUCTION

The purpose of this paper is to discuss one of the most controversial problems arising in distributed databases, namely concurrency control, and to present the various basic techniques which have been proposed to resolve this problem.

Concurrency control is that part of the distributed database management system (DDBMS) which insures that simultaneously executed transactions at multiple sites produce the same results as if they were executed sequentially at a single site. In other words, concurrency control makes transaction data sharing completely transparent to the users. It is easy enough to realize a DDBMS with simplistic though effective concurrency control algorithms such as one which would lock up the whole database during execution of each transaction; such a system would rapidly degrade with increasing number of users and sites so as to become completely unacceptable. Thus we are led to seek simple (i.e. rapidly executing) though effective devices which allow a high degree of concurrency in order to achieve a satisfactory level of performance. It is also quite important to prove correctness of the proposed mechanisms because concurrency control is a tricky problem which has often led to partial or incorrect solutions.

The first goal of this paper is to give a detailed presentation of various concurrency control techniques. The second goal is to introduce certain basic concurrency control principles and to derive from them correctness proofs of the various presented algorithms. In summary, we seek to give a unified view, precise algorithms descriptions and formal proofs of concurrency control techniques. However we do not consider

the multiple copy algorithms although many of these are based on the same principles [GARD80].

The paper is divided into six sections. Section 2 presents certain key definitions concerned with data integrity; it also describes how simultaneous executions of transactions in centralized and distributed database systems may create conflicts which lead to lost operations, currency confusion and database inconsistency. Section 3 analyzes the requirements of conflict-free execution. Having defined a schedule for a set of transactions as a special sequence of the actions making up the transactions, we show that certain schedules introduce conflicts. Thus the goal of concurrency control consists in allowing the execution of solely those schedules that do not lead to such situations. It is shown that this may be accomplished by requiring that both centralized and distributed data management systems generate only serializable schedules, i.e. schedules which yield the same results as a serial schedule. The next question is then how to verify serializability of schedules. It turns out that a sufficient condition for a schedule to be serializable is that its associated precedence graph be circuit-free.

Based on the above principles, three types of approaches have been developed to verify serializability. These are described respectively in sections 4,5 and 6. The first approach consists of two steps: (1) ordering and marking transactions before they are executed, and (2) verifying that conflicting accesses take place in the originally given order; if two transactions are found out of order, one of these is aborted and restarted later. Three types of algorithms have been proposed to accomplish these functions, and these are presented in sections

4.3, 4.4 and 4.5. Total ordering algorithms order all transactions despite the permutability of certain pairs of operations (like read and read). Partial ordering algorithms which only order non-permutable operations are clearly preferable. On the other hand multiversion partial ordering algorithms avoid excessive transaction restarts by maintaining several versions of the data to be accessed. The next approach is derived from the classical operating method for allocating resources to tasks. Thus various portions of data may be viewed as resources which may therefore be allocated (locked) or deallocated (unlocked) to transactions. However, this method by itself is not sufficient here due to the presence of integrity constraints. In order that these be respected it turns out to be necessary to restrict the locking rules so that a transaction cannot unlock any portion of the data until it has locked all the data which it intends to access. These principles are incorporated in a generalized two-phase locking algorithm which we present and prove through the application of precedence graphs. Unfortunately the well-known deadlock problem now appears through data locking. Prevention and avoidance of deadlock is difficult here as these methods require advanced knowledge of the data to be accessed, information that is not available in our context which allows ad hoc queries. Therefore it is necessary to fall back upon deadlock detection. This may be accomplished by algorithm which check for the presence of circuits in the waiting graphs or the allocation graphs of the transaction set of interest. Detected deadlock is then broken by aborting one or more transactions. Another difficulty which may arise is the so-called phantom problem wherein a transactions arrives too late to access a recently inserted data item which it requires; this problem which can occur in

all the methods presented may still require further consideration. The third approach resembles the first although it is based on a new type of transaction ordering, namely the transactions are now ordered according to their termination time. More precisely, transactions are divided into two phases: a process phase and a so-called commit phase during which the database is actually updated. The ordering is here defined at the beginning of the commit phase and, if it turns out that it has been violated during preparation for the actual update, then the transaction is aborted.

Finally, one might be tempted to construct algorithms based on the detection of circuits in precedence graphs, however this is a costly operation and does not appear too promising at this time.

2. DEFINITIONS AND PROBLEM STATEMENTS

2.1. Data Integrity

Let us recall first some basic definitions.

A schema (or database scheme) includes both a description of the data proper and the integrity constraints which qualify that data.

The collection of integrity constraints is said to be self-consistent if it contains no contradictions.

A database instance is said to be consistent if its integrity constraints are self-consistent and its data does not violate the constraints.

There are several types of integrity constraints [DATE77]. First of all, an integrity constraint may bear upon individual data items located at a single site. Such constraints are often expressible through the definitions of the elementary data types. Thus, for example we may have:

- Constraints on domain types

  example: the data item salary is of type integer;

- Constraints on domain range

  example: the data item salary must range between 20,000 and 30,000.

An integrity constraint can also bear upon several data items which may or may not be located at different sites. In general global constraints refer to the former, and local constraints to the latter [PARE77]. Such constraints may define interdata relationships such as the following:

- Functional dependencies [CODD70]

  example: the social security number determines uniquely the name of an employee (SS# EN).

- Multivalued dependencies [FAGI77,ZANI76]

  example: the classes taught by a professor are independent of the names of his children (PROF CLASSES|CHILDREN)

- Existential dependencies

  example: a supplier cannot supply a part which does not appear in the description of all the parts.

- Derived dependencies

  example: the quantity in stock for each part must always equal the number purchased less the number sold.

  The constraint A = B represents a special case of derived dependency called a redundancy constraint. When A and B are located at different sites they are called multiple copies [CHU69].

- Another important type of integrity constraints concerns invariance under transactions. Thus for example, the sum of the balances in the database of a bank must remain constant upon completion of the transaction "transfer".

- Finally it should be noted that certain integrity constraints are time-dependent, thus for example they might only be satisfied at the end of the week or at the end of the month [BENC76].

2.2. Concurrency Problems

2.2.1. Lost Operations

A well-known example of the problems arising from the simultaneous execution of transactions in a centralized system is the lost operations. The most common situation concerns a lost update [ENGL76]. This problem arises from the following sequence of events: during the execution of a transaction T_1 an operation a_i may read a data item X into a buffer location x_1; then the transaction modifies the object in x_1 through a subsequent operation a_j ($j > i$), utilizing the value previously input into x_1. In the meantime, another transaction T_2 has taken place updating the same data item X through the operation a_k which occurs between a_i and a_j. The effect of operation a_k has been lost. This situation is depicted in figure 1.

A similar situation arises from the currency confusion [ENGL76] which takes place when a transaction maintains a reference to a data item destroyed by another transaction.

Lost operations represent a classical problem in centralized systems [GARD77]. The problem is similar in the case of distributed systems [ADIB78]. Here, however the read and write operations which involve remote data require an exchange of messages; furthermore the number of transactions occurring over distributed data is generally greater than in a centralized system. Thus the frequence of lost operations is higher than in a centralized system.

2.2.2. Inconsistencies

Another type of problem is due to the presence of local and/or global integrity constraints. This is the problem of inconsistency [ESWA76] which arises each time a transaction accesses or, worse yet, modifies a transitional state of the database characterized by the fact that an integrity constraint is not verified.

Example 1 (see figure 2):

Let us assume that two data items A and B must satisfy the integrity constraint A = B. Transaction T_2 prints out A after it has been modified by T_1, and B before it has been modified by T_1.

The following example is worse yet for it illustrates the possible destruction of the database.

Example 2 (see figure 3).

Again we assume the integrity constraint A = B. Transaction T_2 now updates a transitory inconsistent state wherein $A \neq B$.

An inconsistency can also be generated by a transaction wherein the same read operation might be repeated with different results [GRAY78]. Indeed a transaction may read twice in a row the same data item yet find two different values due to a modification of the data item caused by a concurrent transaction occurring between the two read operations. This situation is illustrated in figure 4 wherein it may be seen that transaction T_1 prints out two different values for the data item A.

Inconsistency problems are not peculiar to distributed databases as they can already be found in centralized database systems. However they are particularly severe in the former case because of the significant

cost of their resolution which may require numerous messages, lengthy time intervals to reestablish consistency among various distributed databases, and the presence of multiple copies which demand numerous integrity constraints.

In summary, the simultaneous execution of transactions may create conflicts which lead to operation losses or inconsistencies. The problem is to construct control algorithms which only permit the simultaneous execution of those transactions which do not cause such conflicts.

3. CHARACTERISTICS OF CONFLICT-FREE EXECUTIONS

3.1. Basic Concepts

A scheduler controls access of transactions to a portion of the database which is called a data granule or simply a granule.

DEFINITION 1: DATA GRANULE or GRANULE

A data granule or granule is a unit of data access which is individually controlled by a scheduler.

Presently, there are no distributed database systems possessing a single scheduler; this is probably due to performance requirements. Generally there is either a central scheduler which controls through local schedulers access to all the data or a non-hierarchic collection of local schedulers which control access to local data and which are synchronized so as to resolve global conflicts. Thus a data granule is always a collection of centralized data managed by a single site.

The size of data granules is a controversial point [RIES77, RIES79]. It may vary from a single record of a local database to a page or even a whole file. Small granules favor parallelism but require more time for control. In practice it is often desirable to provide variable size data granules to cater more closely to the data needs of various transactions.

DEFINITION 2: ACTION [ESWA76].

A primitive indivisible processing command which is executed on behalf of a single user is called an action.

A granule must obey certain internal integrity constraints. Indeed, during an update of the database, the granules generally are

modified by a number of actions constituting functional units which must respect the internal consistency of granules, i.e. the integrity constraints which qualify the data making up the granules [GARD76, SCHL79].

DEFINITION 3: OPERATION

An operation is a sequence of actions which performs upon a granule a function that respects its internal consistency.

Thus for example, if the granule is a page, then the basic operations are often taken as read page and write page which constitute primitive (indivisible) actions in many systems. When the granule is a record, read record and write record are still basic operations, but then so are modify record and insert record. From these basic operations it is possible to construct other ones such as add (salary, amount) which permits to add the value amount to the attribute salary of the records upon which the operation is performed.

DEFINITION 4: TRANSACTION [ESWA76]

A transaction is a sequence of actions which respect the database integrity constraints, whose execution is triggered by one or more input messages thereby generating one or more output messages.

When a complete local database is taken as a granule, then every local transaction constitutes an operation.

The application of an operation to a granule yields a result.

DEFINITION 5: RESULT OF AN OPERATION

The result of an operation comprises the value(s) created by the operation together with side effects (if any).

Thus for example, the result of the operation read is the value of the input buffer following execution. The result of a transaction modifying a database consists of the modified granules together with the emitted messages.

3.2. Scheduling of Distributed Transactions

In a centralized system, a scheduler controls the simultaneous execution of a set of transactions $\{T_1, T_2, \ldots, T_n\}$. Thus it is convenient to introduce the notion of a schedule for the transactions which has been described as a history or audit [GRAY78].

DEFINITION 6: SCHEDULE

A schedule for a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is a sequence of actions constructed by merging the actions of $T_1, T_2, \ldots, T_n$ while respecting the order of the actions making up each transaction.

A set of transactions can thus give rise to many diverse schedules for these transactions. However each of these schedules maintains the order of the actions making up each transaction. Thus for example, consider transactions T_1 and T_2 depicted in figure 5 which modify data items A and B that are linked by the integrity constraint A = B. Furthermore, we shall assume that A and B are individual granules thereby maximizing the possibilities of concurrency. Two schedules for transactions T_1 and T_2 are shown in figure 6.

As shown in figure 7, a distributed system consists of a collection of local operating systems (OS1, OS2,...) each running on a local processor and processing a collection of transaction programs $TP_{i,j}$ which

appear within various application programs. These local transaction programs may give rise to any number of transactions T_k triggered by end-users' input messages. In addition, any number of subtransactions may be constructed by the local OS in order to access the local database to satisfy a request from a transaction or subtransaction being executed at a remote site.

DEFINITION 7: SUBTRANSACTION

A subtransaction is an action or a sequence of actions carried out at a local site in answer to a transaction (or subtransaction) at a remote site.

Subtransactions are treated like local transactions by the local OS and appear among the collection of T_i's in figure 7. The figure also shows the transaction and an action from that transaction (these being presumably implemented as pointers) currently being executed. The local operating system includes the database manager (DBM), the data communication manager (DCM), and the transaction manager which includes in turn the local scheduler [GARD79] among other modules. The various local schedulers communicate with each other through their data communication managers in order to achieve concurrency control.

It has been demonstrated [TRAI79] that a distributed system executing a set of transactions may be considered as a single system carrying out a schedule of these transactions.

THEOREM 1:

A distributed system executing in parallel a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is equivalent to a centralized system executing some schedule of these transactions.

PROOF

The proof is based on the definition of a global time of the distributed system permitting a total ordering of all executed actions while maintaining the relative order of the actions within each transaction. First we define a local clock at each scheduler which is incremented by one unit every time an action is executed by the local system:

$$CLOCKL(i) := CLOCKL(i) + 1.$$

The problem is now to synchronize the local clocks so as to maintain the order of the actions within each transaction. Let CLOCK(i) denote the synchronized clock situated at site i. One way to define CLOCK(i) is as follows: CLOCK(i) is changed exactly like CLOCKL(i) with the following addition: each time a subtransaction is requested at site i by site j, a message is sent from scheduler j to scheduler i bringing the value of CLOCK(j) at the moment of its emission. Similarly each time a subtransaction is completed at site j on behalf of a (sub)transaction at site i, a message is transmitted from j to i together with the result of that subtransaction. When scheduler i receives such a message it sets its clock to

$$CLOCK(i) := MAX\ (CLOCK(i),\ CLOCK(j)+1).$$

We now assign to each action executed at site i a global time obtained by concatenating the value of CLOCK(i) with the site number i. For example an action executed at site 3 wherein CLOCK(3) = 7, is assigned a global time of 7.3. Thus every transaction executes its successive actions at increasing times. Furthermore two actions cannot be executed at the same time at different sites since their time differs at least in the rightmost digits. Therefore it is possible to define a unique sequence of actions by ordering them according to the global time

assigned to them. This sequence represents a schedule generated by the distributed system which may be utilized by a centralized system to accomplish the same result.

QED

It is clear that, according to the initial value of the local clocks, different schedules may be generated by a distributed system: as we shall see later, all these schedules correspond to the arbitrary orderings which yield the same results.

3.3. Serializable Schedules

As we saw above, certain schedules introduce operation losses or inconsistencies. The problem of concurrency control consists in allowing the execution of solely those schedules which exclude them. It is well-known that a successive execution of transactions (excluding simultaneous transactions) corresponds to a special schedule which has no loss of operation nor inconsistencies. Such a schedule is called a serial schedule and may be formally defined as follows [ESWA76]:

DEFINITION 8: SERIAL SCHEDULE

A schedule S of a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is a serial schedule if there exists a permutation Π of $\{1, 2, \ldots, n\}$ such that $S = < T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(n)} >$.

In order to guarantee the lack of conflict, it is convenient to allow only those schedules which yield the same result as a serial schedule [ESWA76, PAPA77]. Such schedules are said to be serializable.

DEFINITION 9: SERIALIZABLE SCHEDULE

A schedule of a set of transactions $T_1, T_2 \ldots, T_n$ is serializable if it yields exactly the same result as a serial schedule of $\{T_1, T_2, \ldots, T_n\}$.

Thus the problem of concurrency control may be resolved by requiring that a centralized or distributed system generates solely serializable schedules which is a sufficient although not a necessary condition to guarantee lack of conflict [GARD77].

3.4. Properties of Operations

Two operations which do not modify any granules and which belong to two different transactions can always be executed simultaneously without affecting the results of their execution. In other words, every arbitrary scheduling of operations which only perform read actions lead to the same result as that obtained from a sequential scheduling of these operations. It is possible to define more generally the notion of compatible operations.

DEFINITION 10: COMPATIBLE OPERATIONS

Two operations O_i and O_j are compatible if, for any simultaneous execution of O_i and O_j, the result of this execution is the same as that produced by executing O_i followed by O_j or vice-versa.

Consider for example the operations depicted in figure 8. It may be seen that operations O_{11} and O_{21} are compatible whereas O_{12} and O_{22} are not.

It is evident that two operations are noncompatible when there exists a possible merging of operations which generates a loss of

operation. It is also important to note that two operations bearing upon two different granules are always compatible since no loss of operation can be made to occur in this case by interspersing operations.

In certain situations the order in which two operations are performed may affect their result; in others this is not the case. Thus we are led to the notion of permutable operations.

DEFINITION 11: PERMUTABLE OPERATIONS

Two operations O_i and O_j are permutable if every execution of O_i followed by O_j yields the same result as the execution of O_j followed by O_i.

Thus for example operations O_{13} and O_{23} which are depicted in figure 8 are permutable whereas O_{12} and O_{22} are not. It should be noted that two operations bearing upon different granules are always permutable since the execution of the first operation does not affect the result of the second and vice-versa.

3.5. Characteristics of Serializable Schedules

In order to bring out the characteristics of serializable schedules we shall introduce two basic transformations of transaction schedules. First of all the separation of two compatible operations O_i and O_j which are executed by different transactions consist in replacing the simultaneous scheduling of these operations $S(O_i,O_j)$ by a sequence which yields the same result $\langle O_i, O_j \rangle$ or $\langle O_j, O_i \rangle$. The separation of operations thus permits to form a sequence of compatible operations executed by different transactions. Next, the permutation of two permutable operations O_i and O_j which are executed by different transactions con-

sists in exchanging the order of execution of these operations. Thus for example, the sequence $\langle O_i, O_j \rangle$ is replaced by the sequence $\langle O_j, O_i \rangle$.

It is then possible to give a sufficient condition for a schedule to be serializable.

THEOREM 2

A sufficient condition for a schedule to be serializable is that it can be transfomed through separation of compatible operations and permutation of permutable operations into a serial schedule.

PROOF

By definition the results are invariant under separations and permutations. Now, if the schedule can be transformed into a serial schedule, it yields the same result and therefore it is serializable. The condition is not necessary since certain operations which are noncompatible or nonpermutable can be executed simultaneously without conflict at least for special data.

QED

Consider for example the schedule (a) depicted in figure 6. Representing only the operations, this schedule may be written:

T_1 : A + 1 → A

T_2 : A * 2 → A

T_1 : B + 1 → B

T_2 : B * 2 → B.

Operations "A * 2 → A" and "B + 1 → B" are permutable for they bear

upon different granules A and B. Therefore the schedule may be transformed into:

$$T_1 : A + 1 \rightarrow A$$

$$T_1 : B + 1 \rightarrow B$$

$$T_2 : A * 2 \rightarrow A$$

$$T_2 : B * 2 \rightarrow B$$

which is a serial schedule of T_1 and then T_2. Therefore the schedule depicted in figure 6(a) is serializable.

3.6. Precedence Graphs

An important special case arises when the selected granule is a page. Here the two basic operations upon a page of the database are:

(1) READ page

(2) WRITE page.

It may be noted that these operations are quite often seen as atomic (indivisible) actions by the system. Generally, whenever the granule coincides with the unit of transfer between the database and the transaction, then it is possible to consider the operations "READ granule" and "WRITE granule" as basic operations from which all others may be constructed. In order to avoid conflicts the transactional system must insure separability of these basic operations. This is generally done by means of semaphore-like synchronization mechanisms [DIJK68,COUR71] permitting exclusion of READ/WRITE operations on the same granule. Henceforth and unless stated otherwise, we shall therefore consider the types of systems found in practice wherein there are two basic operations "READ granule" and "WRITE granule" which are always separated when they bear upon the same granule.

In practice, separation followed by permutation of operations bearing upon different granules is always possible. Indeed only certain permutations of operations bearing upon the same granules are impossible. More precisely, if g is a granule, we have the following:

(1) (T_i : READ g) and (T_j : READ g) are permutable operations

(2) (T_i : READ g) and (T_j : WRITE g) are non-permutable operations. Therefore, when T_i reads a granule before T_j writes it, we say that T_i precedes T_j; this precedence relation is denoted $T_i < T_j$ [BERN80].

(3) (T_i : WRITE g) and (T_j : WRITE g) are non-permutable operations. Therefore, when T_i writes a granule before T_j writes it too, we say again that T_i precedes T_j, denoting this again $T_i < T_j$.

The notion of precedence permits to define a graph [TRIN75, BADA80, BERN80] which characterizes the possible relationships among transactions during scheduling.

DEFINITION 12: PRECEDENCE GRAPH

A precedence graph of a schedule is a graph whose set of vertices is the set of transactions and such that there exists an arc from T_i to T_j if T_i precedes T_j ($T_i < T_j$).

It is now possible to demonstrate the following theorem:

THEOREM 3

A sufficient condition for a schedule to be serializable is that its associated precedence graph have no circuit.

PROOF:

Let us consider a schedule S with a circuitless precedence graph. This precedence graph represents precedence relations among transactions due to non-permutable operations. With the circuitless precedence graph it is possible to define among the transactions a partial order which respects precedence relations due to nonpermutable operations. Such a partial order may be extended arbitrarily into complete ordering $<T_{i1}, T_{i2}, \ldots, T_{in}>$. All the other operations being permutable, they may be rearranged to fit this order.

QED

An application of theorem 3 is shown in figure 9 which depicts the precedence graphs corresponding to the schedules shown in figure 6. Graph (a) has no circuit, thus confirming the fact that schedule (a) is serializable. On the other hand graph (b) has a circuit, thus explaining the fact that schedule (b) is not serializable.

4. ALGORITHMS BASED ON THE INITIAL ORDERING OF TRANSACTIONS

4.1. Transactions Timestamps

A first method to prevent the production of nonserializable schedules consists in ordering the transaction when they are launched into execution and to insure that conflicting accesses to granules take place in the transaction given order. Such a method is the basis of concurrency control in SDD.1 [BERN80]. To this end it is necessary to associate with each transaction a unique identifier called a transaction timestamp [ROSE78] making it possible to verify their ordering.

DEFINITION 13: TRANSACTION TIMESTAMP

A transaction timestamp is a numerical value assigned to a transaction which allows it to be ordered with respect to other transactions.

In a distributed system the generation of transaction timestamps is not a trivial problem. A first method consists in utilizing a global time [LAMP78] such as that defined in the proof of theorem 1. This time consists of the local time concatenated with the site number in the least significant digits and synchronized with the local clock that is ahead. Finally the device for defining global time which was formally described in the proof of theorem 1 permits to assign unique transaction timestamps to the transactions still respecting the temporal order of their activations. No additional messages are required. However, the size of the timestamp may be inconvenient since local time plus site number may require several bytes (4 to 8).

A second method has been proposed by project SIRIUS [LELA78]. It consists in circulating through the network a unique sequencer. As described in the original version, each time a site wishes to launch a transaction, it awaits the sequencer and assigns the number that the sequencer carries to that transaction. The sequencer is then incremented by one and sent to the following site. In order that every site may access periodically the sequencer, the latter loops over a virtual ring linking all the sites. To avoid excessive delays while waiting for the sequencer, a site may, when accessing it, reserve several consecutive numbers from the sequencer which it may assign later as transaction timestamps until the next passage of the sequencer. One of the problems of this method is the necessity to regenerate the sequencer when the site which holds it fails. This may be achieved by means of a virtual ring and a sequencer regeneration device at each site. Whenever a site is detected as having failed, it is eliminated from the virtual ring which is then regenerated. Whenever a site has not received the sequencer within a certain time interval, it sends to the next site over the virtual ring the last sequencer value it had received. Thus several sites may regenerate the sequencer, but only the largest value must be preserved, the others being eliminated within the first circuit [LELA78]. The advantage of the sequencer mechanism lies in the relatively small size of the generated timestamp (16 bits). The disadvantages are (1) that it requires a permanent communication among all the sites thus restricting its practical utilization to local networks and (2) the complexity of regeneration when a site fails.

4.2. Granule Timestamps

In order to insure that the transactions operate upon the granules in the order defined by the transaction timestamps it is necessary to remember the timestamp of the last transaction having operated upon a granule. This is done by means of the granule timestamps [THOM79].

DEFINITION 14: GRANULE TIMESTAMP

A granule timestamp is a numeric variable associated with a granule which stores the timestamp of the last transaction having operated upon that granule.

In order to provide a better characterization of the operations being carried out, it is possible to distinguish between a read granule timestamp (value of the transaction timestamp of the last transaction having done a READ on that granule) and a write granule timestamp (value of the transaction timestamp of the last transaction having done a WRITE on that granule).

The processing of granule timestamps as a means for controlling concurrent access may be carried out by reserving storage space in each granule. Such a strategy has two disadvantages:

(1) the amount of utilized storage space may be non-negligible, for example several bytes per page;

(2) updating the granule timestamp may demand additional i/o operations, especially during read operations since these may require updating the granule timestamp on the disk.

These problems may be circumvented by "forgetting" timestamps which are too old, utilizing the following techniques:

(1) Since access conflicts can only occur among current transactions, it is not necessary to preserve the timestamps corresponding to transactions which are not simultaneous with current transactions. Assuming T is the maximum duration of a transaction and t is the current global time, then any granule timestamp smaller than t - 2T may be discarded for it corresponds to a transaction necessarily completed at the time of initialization of the oldest current transaction.

(2) Granule timestamps can also be stored in limited size tables of the central memory [BERN80]. An entry in such a table consists of the granule identifier together with the corresponding granule timestamp. In addition to this table one also needs a variable containing the value m of the largest granule timestamp value which has been expurgated from the table. In order to determine a granule timestamp value, the scheduler seeks that granule in the table. If that granule is found its timestamp is also found in the table, otherwise the value of m is taken as the value of the timestamp of the sought granule. This insures secure controls regardless of the size of the table and of the strategy utilized to eliminate timestamp from the table since the actual value of the timestamp of the sought granule is necessarily smaller or equal to m.

4.3. Total Ordering Algorithms

This algorithm consists in verifying that transactional access to the granules takes place in the order initially assigned to transactions and indicated by the transaction timestamp. Whenever two transactions, T_i having timestamp i and T_j having timestamp j where $i < j$, operate

upon the same granule the scheduler controlling access to this granule makes sure that T_i precedes T_j. If this is not the case the scheduler simply aborts one of the transactions which will be restarted later. It must be emphasized that this algorithm does not distinguish among read and write operations and thus orders unnecessarily read operations which are permutable and therefore nonconflicting. On the other hand it needs only one type of granule timestamp.

Figure 10 depicts the total ordering algorithm. Here g denotes the granule accessed by the transaction T_i having transaction timestamp i. S(g) is the granule timestamp of granule g. ABORT is a procedure causing restart of transaction T_i or T_j.

The problem of restarting transactions following ABORT can be treated differently depending whether the aborted transaction is T_i, the one which requests an operation, or T_j, the one which has already performed the operation which led to the conflict. In the first case T_i must be restarted with a new transaction timestamp i' greater than j in order to eliminate the conflict with T_j. However, T_i, may now be in conflict with a new transaction, say $T_{j'}$. Thus T_i, may be aborted again and this process may continue indefinitely. One might try to halt these repeated aborts by assigning arbitrarily to the aborted transaction a sufficiently large transaction timestamp [BERN80], however another aborted transaction may still create repeated conflicts.

In the second case, T_j may be restarted with the same transaction timestamp. Indeed an older transaction (having smaller transaction timestamp value) takes priority therefore there is no danger of continuous restart of the transactions. However, a difficult (probably

insoluble) problem arises in this method: a restart of T_j requires (1) the restoration of the preceding version as well as that of the granule timestamp of the accessed granule, and (2) reverification that the order of the access respects the order implied by the transaction timestamp; if this is not the case it is necessary to restart the transaction preceding T_j. Note that this implies the need to preserve several versions of a granule (we shall examine later in detail various algorithms to this effect). The other problem is that the device may cause a restart of completed transactions. In order to avoid this difficulty it is necessary to declare a transaction as terminated only when it cannot be aborted, i.e. when all older transactions are completed. This leads to ordering the transactions commit according to the values of their transaction timestamp. In other words aborting a transaction T_i may lead to a situation where an earlier update of T_i is undone while its effect remains visible on a subsequent transaction say T_j. This may require an abort of T_j which in turn may lead to the abort of yet another transaction etc.. Such a set of circumstances has been called the domino effect in another context [MENA78].

4.4. Partial Ordering Algorithms [BERN80]

The preceding algorithm orders all operations on granules. Actually it is only necessary to order nonpermutable operations, i.e. READ/WRITE and WRITE/WRITE. The algorithm we present here consists in verifying that the sequences <READ, WRITE>, <WRITE, READ> and <WRITE, WRITE> are performed in the order assigned initially and defined through the transaction timestamps. To do this, two types of granule timestamps are utilized: SR, the read granule timestamp which is the timestamp of

the most recent transaction (the transaction whose timestamp has the greatest value) having executed a READ operation; and SW, the write granule timestamp which is the timestamp of the last transaction having executed a WRITE operation. When a transaction executes a READ, the scheduler controls the correct sequence of the READ with respect to the last WRITE, i.e. the timestamp of the reading transaction must be greater than the value of the write granule timestamp. When a transaction executes a WRITE, the scheduler controls the correct sequence of the WRITE with respect to the previously executed READ and WRITE, i.e. the timestamp of the writing transaction must be greater than the value of the write granule timestamp and that of the read granule timestamp. Figure 11 shows the partial ordering algorithm. Apart from the granule timestamps of the granule g which are denoted SR(g) and SW(g), the notation is the same as that of figure 10.

It must be realized that the partial ordering algorithm suffers from the same problems as those afflicting the preceding total ordering algorithm.

4.5. Multiversion Partial Ordering Algorithms [BERN80]

The strategy described above may be improved by preserving several versions of the same granule. For each granule g, the system should preserve:

- a set of write granule timestamps $\{SW_i(g)\}$ with the associated granule data value $\{v_i(g)\}$;
- a set of read granule timestamps $\{SR_i(g)\}$.

Each of these correspond to a granule version i.

It is then possible to insure the proper order of read operations with respect to write operations without ever restarting a read transaction. To do this it suffices to provide transaction T_i requesting to read granule g with the version whose write granule timestamp is immediately below i. Thereby T_i shall always precede all transactions of higher timestamps writing upon the granule in questions, and shall follow all transactions with lower timestamps. Therefore T_i will be in proper order. Everything will take place as if T_i had requested to read just after the write of the version having immediately lower granule timestamp. The control algorithm for the READ operation including a device for multiversions partial ordering is depicted in figure 12.

It is also possible to force an ordering of the writing operations of T_i by inserting a new version created by T_i just after that having a granule timestamp immediately below it, say v_i. If, however, before the T_i write actions, a transaction T_k having a timestamp higher than that of T_i ($k > i$) has read the version v_j, then either T_k or T_i must be aborted. However, T_k can only be aborted if it is not yet terminated. Therefore it is preferable to restart T_i with a new timestamp i' greater than k. These two possibilities are illustrated in figure 13 with $i = 6$, $k = 7$ and $j = 5$. The control algorithm for the operation WRITE is presented in figure 12. The notation is the same as in figure 11 with the addition of an index representing the various versions.

5. TWO-PHASE LOCKING ALGORITHMS

5.1. Basic Principles

The previous strategies permit execution of transaction while verifying that conflicting accesses to the same granule are executed in the order initially defined when launching the transactions. If this order is not respected one of the conflicting transactions is generally restarted. Therefore these strategies may be described as detecting non-serializable schedules which are corrected by restarting certain transactions. In contradistinction, locking type strategies consist in avoiding the generation of incorrect schedules by delaying one of the transactions which attempt to execute conflicting operations on the same granule.

The prime goal of locking algorithms is to allow simultaneous execution of only those operations which are compatible. This requires the introduction of the concept of operation mode.

DEFINITION 15: OPERATION MODE

An operation mode is a property which characterizes an operation by permitting to determine its compatibility with other operations.

The most common operation modes are the read and update modes. The read mode characterizes every operation which performs read actions only whereas the update mode corresponds to every operation which performs a write, insert or modify action; it also allows read actions. On the other hand, the DBTG [CODA71] has proposed the following modes:

M1 = nonprotected retrieval,

M2 = protected retrieval,

M3 = nonprotected update,

M4 = protected update,

M5 = exclusive retrieval,

M6 = exclusive update.

Generally it is possible to determine the compatibility of these modes through a matrix C called the compatibility matrix. This matrix is defined as having elements $C_{ij} = 1$ iff the operation modes M_i and M_j are compatible, and $C_{ij} = 0$ otherwise. It should be noted that the matrix C is symmetrical since the compatibility relation among modes is itself symmetric. The compatibility matrix for the above mentioned read-write modes (m_0 = read mode, m_1 = write mode) is defined by

$$C = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

The matrix for the DBTG modes is as follows:

$$C = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

5.2. Locking Protocols

A scheduler should only permit simultaneous execution on the same granule of operations having compatible modes. To do this, it must know when an operation starts on a granule as well as the operation modes

required to carry out the given operation. Similarly, it is necessary to indicate to the scheduler controlling a granule that an operation is completed. To accomplish all this, two special actions are introduced:

- LOCK (g,M) which permits a transaction to signal to the scheduler controlling granule g the start of an operation of mode(s) defined by M;
- UNLOCK (g) which permits a transaction to signal to the scheduler controlling granule g the end of the current operation bearing on g.

The proposed protocol requires the execution of two additional actions per operation: LOCK and UNLOCK. The interface seen by the transactions may be simplified by combining automatically a locking request with the actions permitting to search for a granule in the database. The DBTG recommendations [CODA71] permit for example to lock automatically the current record associated with a transaction and to unlock it as soon as it is no longer referenced by a currency indicator. Two more primitives have also been proposed to permit locking other than current records (KEEP and FREE). Many systems insert the unlocking operation either at the end of transactions or at certain intermediate points where the database is consistent. Thus it is no longer necessary to request unlocking of each granule. This also facilitates restarts in case of failures. Whatever be the case, even if the interface LOCK/UNLOCK is invisible to the users, there exists at least an equivalent logical interface within the system.

5.3. Locking Algorithms

In order to permit simultaneous execution on the same granule of compatible operations only, it is necessary to record the modes of the operations currently bearing on every granule. This may be simply achieved by associating to each pair <granule g, transaction T_i>, operating on the granule, the bit vector

$$A(g,i) = \begin{bmatrix} a_i \\ \vdots \\ a_j \\ \vdots \\ a_k \end{bmatrix}$$

where $a_j = 1$ if an operation of mode M_j is executing on behalf of transaction T_i upon granule g, and 0 otherwise. Similarly, we assume that the modes requested by the action LOCK(g,M) are defined through a bit vector

$$M = \begin{bmatrix} m_i \\ \vdots \\ m_j \\ \vdots \\ m_k \end{bmatrix}$$

where $m_j = 1$ if the mode M_j is requested, and 0 otherwise. It is now possible to prove a theorem where the following operators appear:

- $\vee$ logical union of two boolean vectors (logical OR),
- $\wedge$ logical intersection of two boolean vectors (logical AND),
- $\neg$ logical negation of a boolean vector,
- $\subset$ logical inclusion of a boolean vector,
- $*$ boolean matrices product (this corresponds to the usual matrix product where the add operation is replaced by logical union and the multiply operation is replaced by logical intersection).

THEOREM 4:

The operation modes requested during a primitive operation LOCK(g,M) executed by a transaction T_p are compatible with the modes of the operations which are currently executing upon granule g through the other transactions if:

$$M \subset \neg(\neg C * \bigvee_{i \neq p} A(g,i)).$$

PROOF:

$\bigvee_{i \neq p} A(g,i)$ is a bit vector whose j bit is 1 if M_j is the mode of an operation currently being executed upon granule g by a transaction other than T_p. $\neg C$ is the complement of the compatibility matrix. Therefore,

$\neg C * \bigvee_{i \neq p} A(g,i)$ is a bit vector whose j bit is 1 if the mode M_j is incompatible with the mode of an operation currently being executed on granule g by a transaction other than T_p. Finally $\neg(\neg C * \bigvee_{i \neq p} A(g,i))$ is therefore a bit vector representing all the operation modes which are compatible with the modes of operations currently being executed on granule g by a transaction other than T_p. Therefore, every vector included within the latter corresponds to modes which are compatible with those of operations currently being executed on granule g by a transaction other than T_p.

QED

A locking algorithm requires the definition of a strategy for the case when the requested operation modes during the execution of a primitive LOCK are not compatible with the modes of currently executed

operations. The simplest strategy consists in returning the message "granule busy" to the transaction. In order to control the waiting stages, it is generally preferable to place the request into a waiting state and block the calling transaction until the requested granule becomes available. A waiting queue Q[g] can then be associated with each busy granule. This queue contains the locking requests (consisting of the vector M and the transaction identifier) which are queued in order of priority, this coinciding generally with the order of their arrival. Figure 14 depicts the locking/unlocking algorithm corresponding to such a strategy for the transaction T_p.

5.4. Two-phase Restriction [ESWA76]

The locking algorithm presented above allows the simultaneous execution on the same granule of compatible operations only. It has been shown that this may be achieved by restricting the usage of the actions LOCK and UNLOCK in the transactions [ESWA76]. We shall now introduce this restriction which is called the "two-phase restriction".

First of all, it is necessary to emphasize that every transaction which respects the locking protocol as defined up to here, must execute LOCK with the correct operation mode before executing an operation on a granule. It must also execute UNLOCK after the end of the execution of this operation. Henceforth, we shall assume all the transactions possess these properties of well-formedness.

We can now introduce formally the notion of two-phase transaction.

DEFINITION 16: TWO-PHASE

A transaction is two-phase if it does not execute a LOCK after it has executed an UNLOCK.

Thus, a two-phase transaction must lock all the granules upon which it operates before it begins to unlock. A typical curve depicting the variation in the number of granules locked by a two-way transaction is shown in Figure 15.

The following theorem shows the importance of two-phase transactions and shall permit us to introduce the user rules of the locking protocol which only allows the production of serializable schedules.

THEOREM 5.

Every schedule of a set of two-phase transaction is serializable.

PROOF:

A serializable schedule possesses a precedence graph with no circuits. Let us consider then a schedule for the set $\{T_1, T_2 \ldots, T_n\}$ of two-phase transactions, and let us assume that there exists a precedence circuit $T_{i1} < T_{i2} < \ldots < T_{in} < T_{i1}$. It follows immediately that

- T_{i2} locks (through LOCK action) a granule g_{i1}, after T_{i1} unlocks (through the UNLOCK action) that granule;
- T_{i3} locks a granule g_{i2} after T_{i2} unlocks that granule;

.....

- T_{il} locks a granule g_{in} after T_{in} unlock that granule.

Since each transaction T_{i2}, T_{i3},... is two-phase, it unlocks only after it has completed all its LOCK operations. Thus, T_{il} unlocks g_{il} before executing the lock of g_{in}. Therefore, T_{il} is not two-phase which contradicts the initial hypothesis.

QED

According to Theorem 5 and the preceding remark, it is possible to introduce user rules for the primitives LOCK/UNLOCK in a distributed (or centralized) database:

R1: Every transaction must execute LOCK with the correct operation mode(s) upon the chosen granule before executing an operation on this granule;

R2: Every transaction must execute UNLOCK on the selected granules after execution of the operation;

R3: A transaction cannot execute LOCK after executing UNLOCK.

Every scheduler must verify that the transactions whose schedule it controls do follow these rules. In the common case where the operation modes for reading and writing are distinct, rule R1 requires that a transaction request a read lock when reading a granule which it will not modify, and a write lock when reading a granule which it will modify.

Finally, two-phase locking requires that each transaction pass through a state of maximum locking wherein all the accessed granules are locked. It follows that the accessing order to the granules forced upon the transactions is that one in which the transactions reach this max-

imum locking state. Two-phase locking may therefore be considered as an ordering of the transactions. This order is not constructed a priori as in the algorithms based upon initial ordering, instead it is determined by the point in time when the transactions reach their state of maximum locking.

5.5. Locking in a Distributed System

The application of two-phase locking in a distributed system requires a selection of one or more sites for executing of LOCK/UNLOCK algorithms More precisely, since the granules are distributed over several computers, the problem is to determine who is to manage the lock states of the granules and the queues of locking requests. Two types of solutions have been widely studied: centralized control and distributed control [ALSB76, ELLI77].

Centralized control is realized when a single site is in charge of locking/unlocking. The site is provided with a centralized controller which controls access to the data. When a transaction demands to lock a granule, the associated scheduler transmits the locking request to the central controller which then executes the locking algorithm. If the granule is busy, two strategies are possible: either a message of refusal to lock is returned to the requesting scheduler which shall then be able to repeat the request later, or the request is filed into the queue until the busy granule is unlocked. The second strategy is clearly less costly in terms of transmitted messages, but it may be difficult to realize because of the complexity of the mechanism required to handle messages without immediate response. In any case, the centralized con-

trol system suffers from the major disadvantage that when the central controller breaks down the whole operating system for the distributed database grinds to a halt. A solution which is breakdown-proof [MENA80] consists in informing several sites of the states of allocation of the granules and then selecting a new central controller when the current one breaks down. Such a solution carries a high message cost and tends toward a decentralized control.

Distributed control is realized when each scheduler is responsible for locking/unlocking of the granules which are managed by the local associated DBMS. In such a method there is therefore a controller at each site whose assignment is to lock/unlock the local granules for all transactions. When a transaction demands to lock a granule the schedules controlling the transactions must therefore transmit the LOCK request to the scheduler controlling the granule in question. As in centralized control, and with the same advantages and disadvantages, the waiting queues may be processed by the scheduler controlling the granule. Finally, distributed control offers the major advantage that it allows to continue operation of the distributed system even when some of the computers have broken down: control of the system no longer depends upon a unique computer.

5.6. The Deadlock Problem

Deadlock, or deadly embrace, is the situation which arises when granules have been locked in such a sequence that a group of transactions has the following two properties [COFF71]:

(1) Every transaction of the group is blocked, waiting for a granule;

(2) The execution of every transaction which does not belong to the group does not permit to unblock any transaction in the group.

Deadlock is a problem generated by locking which is difficult to resolve. We shall examine below the main solutions which have been proposed first in a centralized and then in a distributed system.

5.6.1. Representations of deadlock

First, we shall present an example of deadlock. Let us consider two transactions, T_1 and T_2. Transaction T_1 has succeeded in locking granule G_1 for update and is waiting to lock granule G_2 also for update. Transaction T_2 has succeeded in locking the same granule G_2 for update and is waiting to lock G_1 for protected retrieval (see Figure 16).

The impossibility of executing simultaneously updates with other updates or protected retrievals upon the same granules results in the two transactions waiting for each other: it is a deadlock situation.

We present below two representations of deadlocks in terms of special graphs called the waiting graph and the allocation graph.

The waiting graph [MURP68] is a graph $G(\mathcal{T}, W)$ where $\mathcal{T}$ is the set of concurrent transactions $\{T_1, T_2, \ldots, T_n\}$ sharing granules $\{G_1, G_2, \ldots, G_m\}$, and W is the relation "waits" defined as follows:

T_p waits for T_q if T_p demanded to lock a granule G_i and this request cannot be accepted because G_i has been locked by T_q.

Murphy has presented in 1968 the following theorem:

THEOREM 6:

There exists a deadlock if the waiting graph has a circuit.

PROOF:

If the waiting graph has a circuit, then there exists a group of transactions wherein:

T_1 awaits T_2

T_2 awaits T_3

&.....

T_k awaits T_1.

Every transaction in the group is therefore blocked waiting for a granule because this granule is utilized by another transaction of the group. Therefore, completion of all the transactions which are not included in the group does not allow unblocking any transaction of the group.

Vice versa, the presence of deadlock implies the presence of at least one circuit. Otherwise every group of transactions would be such that the waiting subgraph which it generates has no circuit. Following execution of all the transactions which do not block the group, it would therefore be possible to unblock a transaction of the group since a graph without circuit has at least one terminal vertex.

QED

Figure 17 illustrates the application of Theorem 6.

All the transactions belonging to a circuit are deadlocked. Furthermore, a transaction which is waiting for deadlocked transaction is itself deadlocked (see figure 18).

It is interesting to establish the relationship between the waiting graph and the precedence graph. By definition, when a transaction T_i waits for a transaction T_j, this means that T_j has locked a granule G which T_i is requesting in an incompatible mode. Therefore the operation for which T_j has locked G should be performed before that requested by T_i since the two operations are incompatible and thence nonpermutable. Thus T_j precedes T_i with respect to G, i.e. $T_j < T_i$. However the precedence relation does not generally imply the "wait" relation. Therefore, reversing the arcs of the waiting graph yields a subgraph of the precedence graph. This implies that, if the waiting graph has a circuit, then so does the precedence graph. Furthermore a deadlock cannot lead to a serializable schedule even if it was possible to complete the deadlocked transactions.

We now introduce the allocation graph which consists of two sets of vertices:

- The set T of transactions
- The set G of granules.

An arc links granule G_i to transaction T_p if T_p has succeeded in locking G_i in at least one operation mode. An arc links transaction T_p to granule G_i if T_p has requested though not yet obtained locking of that granule. These arcs are colored (i.e. marked) according to the mode of the requested operation. Figure 19 depicts the allocation graph for the example shown in figure 17.

The following theorem is easily proven:

THEOREM 7:

A necessary condition for deadlock is the existence of a circuit in the allocation graph. This condition is generally not sufficient.

PROOF:

We shall demonstrate that if there are no circuits in the allocation graph, there cannot be any deadlock. Let T be an arbitrary set of transactions. Since the allocation graph has no circuit, the subgraph obtained after executing the transactions which do not belong to T has also no circuit. Therefore it has a terminal vertex. This vertex cannot be a granule since an unlocked granule cannot be waited for. Thus, for every set of transactions T, the execution of all the transactions which do not belong to T would unblock a transaction belonging to T. Therefore there is no deadlock.

A simple example indicated that the condition is not sufficient. Let us consider three transactions T_1, T_2, T_3 sharing two granules G_1 and G_2, and utilizing the operation modes of retrieval, update and insertion. Let us also assume that retrieval is compatible with retrieval and insertion, while update-update, update-insertion and insertion-insertion are incompatible. The allocation graph depicted in figure 20 has a circuit although there is no deadlock as shown by the corresponding waiting graph which appears on figure 21. We might note however that the condition is sufficient in the special case where the only modes are read and write as illustrated in figure 19.

QED

It may be noted that in the general case there is no straight forward relationship between precedence graphs and allocation graphs. However, as stated previously, when the only existing operation modes are read and write, the presence of a circuit in the allocation graph is a necessary and sufficient condition for deadlock just as the presence of a circuit in the waiting graph. This in turn implies the presence of a circuit in the precedence graph.

5.6.2. Detection of deadlock in a centralized control system

A deadlock detection algorithm may be constructed from a circuit detection algorithm for waiting graphs. We shall utilize here an algorithm which tests a graph for circuits by successive elimination of terminal vertices [HOLT72].

In a waiting graph, a vertex is terminal if the transaction that it represents is not waiting to lock any granules. Thus, let N(k) be the number of granules that the transaction T_k is waiting to lock. An initial reduction of the graph may be obtained by eliminating its terminal vertices, i.e. those for which N(k) = 0. The problem is then to recompute the values of the N(k)'s following the reduction in order to carry out the next reduction. This may be done by counting the requests which can be satisfied after each reduction, decrementing N(k) for each such request of transaction T_k. This method has two special requirements:

- it is necessary to mark the counted requests in order not to count them more than once;
- it is necessary to utilize a special procedure to test whether a request is satisfied taking into account the locking states of the

transactions which have not yet been eliminated from the waiting graph.

Therefore, let SLOCK (G_i, R, k) be a boolean procedure which tests whether request R of transaction T_k for granule G_i can be satisfied taking into account the locking of granules by the transactions which still remains in the waiting graph. This procedure yields true if the request can be satisfied, and false otherwise. The program for the procedure is analogous to that of the LOCK algorithm except that the locking states are not modified. Figure 22 depicts a procedure DETECT which yields the answer true if there is deadlock and false otherwise.

Generally speaking there are three ways to deal with deadlock: prevention, detection and avoidance [HOWA73]. Prevention and avoidance require advanced knowledge of the granules which will be requested by transactions. However this is not possible for a general ad hoc query system. Thus the only acceptable solution to the deadlock problem in the present context consists in detecting deadlock and, if it is present, restarting the transaction so as break the circuits of the waiting graph.

When deadlock is detected, the problem is to select a transaction for restart. The detection algorithm presented below yields the list of deadlocked transaction. One of these must therefore be selected and restarted. However all solutions are not equally satisfactory as indicated by figure 23 wherein it appears that restarting T_3 does not resolve the deadlock.

A possible solution to this problem may be to restart the transaction which blocks the greatest number of transactions, i.e. that one

which corresponds to the vertex having the greatest indegree in the waiting graph.

In order to reduce the cost of detection with restart it is advisable to invoke the detection algorithm only when a transaction has been awaiting a lock for a long time (say a few seconds), rather than inkoving the algorithm whenever a transaction just started waiting for a lock.

5.6.3. Detection of deadlock in a distributed control system

In a distributed database system, one might believe it sufficient to resolve deadlock problems through the local system among subtransactions and local transactions. This is not the case however for transactions being executed at a given site may be awaiting messages from subtransactions being executed at another site [GOLD77] thereby creating intersite deadlocks.

This problem is illustrated in figure 24 which depicts the following situation:

- At site S1, subtransaction T1 is waiting for a granule allocated to transaction T2. T2 is waiting for a message from its subtransaction T3;
- At site S2 subtransaction T3 is waiting for a granule allocated to transaction T4. T4 is waiting for a message from its subtransaction T1.

5.6.3.1. Centralized execution of the detection algorithm

We shall call centralized execution the solution which consists in executing the deadlock detection algorithm on a computer of the network providing as input the locking states and the waiting requests for shared granules. This may be accomplished in at least two ways

- either before executing the algorithm the computer triggers a search for the various waiting requests and locking states through a special protocol;
- or the computer receives periodically, or possible whenever there is a change, the various conditions of the states [STON78].

The computer which detects deadlock may be a specialized machine although any computer can execute the algorithm.

It should be noted that deadlock detection by a computer of the network does not prevent simultaneous allocation of granules by other computers of the same network. Thus it is possible that the information provided by the detection algorithm may be obsolete; however this is not important since deadlock is a terminal state.

5.6.3.2. Construction of the waiting graph

A solution which requires no more messages than the number of computers in the network can be obtained by explicit construction of the waiting graph. When a computer wishes to verify the presence of deadlock, it collects parts of the waiting graph from each computer of the network. These graph portions may be represented by a boolean matrix $G = (g_{ij})$ where $g_{ij} = 1$ if transaction T_i awaits transaction T_j and $g_{ij} = 0$ otherwise. By union of the various G matrices the

collecting computer can obtain the complete matrix representing the waiting graph. It is then possible to invoke a circuit detection algorithm [MURP68].

5.6.3.3. Inquiry initiation

Several solutions have been proposed based upon distributed detection of circuits in graphs. We present here one of the most efficient ones [GOLD77] which may be considered as a distributed algorithm for detecting circuits in allocation graphs. When a transaction T_0 awaits a granule G (possibly for a long time) and this granule is under the control of transaction T_1, an inquiry is sent to the scheduler where T_1 was initiated. This inquiry is accompanied by a list of objects $<T_0, G_1, T_1>$.

When a scheduler receives an inquiry together with a list of objects terminating with a transaction such as $<T_0, G_1, T_1, G_2, \ldots, T_{i-1}, G_i, T_i>$, it performs the following actions:

(1) verify that T_i has indeed been initiated by this scheduler and that G_i is indeed locked by T_i; if yes, go to (2); if no, stop the inquiry;

(2) if T_i is not waiting for any granule, stop the inquiry; if not, go to (3);

(3) obtain the identifier of the granule awaited by T_i, say G_{i+1}; add G_{i+1} to the list of objects and send an inquiry accompanied by the list of objects $<T_0, G_1, T_1, G_2, \ldots, T_{i-1}, G_i, T_i, G_{i+1}>$ to the scheduler which controls G_{i+1}.

When a scheduler receives an inquiry together with a list of objects terminated by a granule identifier, such as $<T_0, G_1, T_1, G_2, \ldots, T_{i-1}, G_i, T_i, G_{i+1}>$, it performs the following actions:

(1) verify that G_{i+1} is indeed awaited by T_i, otherwise stop the inquiry; if yes, perform (2) for each transaction T_{i+1} which controls G_{i+1};

(2) if T_{i+1} is in the list of objects, there exists a circuit in the allocation graph and thus there is deadlock. The inquiry is therefore terminated successfully. If not, go to (3);

(3) Add T_{i+1} to the list of objects and send the inquiry to the scheduler by which T_{i+1} has been launched.

An evaluation of such a solution is difficult. It is claimed [GOLD77] that the number of messages is small because the inquiries generally stop very quickly when there is no deadlock. It remains to demonstrate that such a solution is less costly than a simpler one wherein the transactions which await a granule beyond a fixed time are restarted.

5.7. Other Problems Due to Locking

Another problem due to locking is that of "starvation" [COUR71] also called permanent blocking [HOLT72]. This problem appears as soon as a group of transactions effectively behave as a coalition by carrying out mutually compatible operations (such as read) in the presence of an individual transaction which intends to execute an operation that is incompatible with the former ones (for example write). The individual transaction may then have to wait indefinitely. The solution to this

problem is the same in the distributed as in a centralized system. Generally it consists in lining up requests for locks in order of arrival, though letting through those which are compatible with all the requests that are ahead including the one that is currently executing.

The "phantom" problem has also been raised in [ESWA76]. This problem comes up when a granule is introduced in the database too late to be processed by a continuing transaction although it affects that transaction later. Consider for example an aircraft flight database shown in figure 25 which consists for each flight of a passenger list and some general flight information including the number of passengers having received a seat on the flight. Let us assume now the following transactions:

T_1 (part 1): list the passengers names and numbers.

T_1 (part 2): list the flight information including flight number and the number of passengers having reserved seats.

T_2: add a passenger <TOPPER, 13> and increment the number of passengers in the flight information.

Let us further assume that these transactions are executed in the following order: T_1 (part 1), T_2, T_1 (part 2). This is a legal schedule since T_2 accesses an unlocked granule which does not even exist when T_1 performs the first part of its transaction. However the result of T_1 is a list of three passengers while the number of passengers is printed as four.

This problem may be resolved by defining logical granules (in the above example this could be the logical relation PASSENGER) through special predicates [ESWA76]. Locking through predicates also permits to

define granules of variable size constructed according to the needs of various transactions. Unfortunately it necessitates special algorithms to determine whether two predicates are disjoint and this problem does not seem to have an efficient solution for generalized predicates. It should also be noted that the phantom problem is not peculiar to locking strategies and can also appear in the transaction ordering approach.

Another problem is the need for granules of variable size thus permitting locking at various levels such as page, file or even database. In order to insure compatibility among various levels, [GRAY78] have introduced the notion of intention mode. Within such a context every granule can operate in two types of modes: normal modes and intention modes which correspond to the intent to lock a subset of the granule under consideration. It has been shown [GARD78] that it is possible to define compatibilities among normal and intention modes starting from p initial modes through the following compatibility matrix:

$$C' = \begin{pmatrix} C & C \\ C & E \end{pmatrix}$$

where the first p modes of operations are the normal modes, the next p modes are the intention modes and E is a matrix consisting of 1's. In order to lock a granule which is in mode j, the algorithm consists in locking all the granules containing the former in intention mode p+j, then locking the granule in normal mode. Such an algorithm is particularly suitable to locking within hierarchies [ULLM80]. It may be utilized in a centralized as well as a distributed system.

6. OTHER CONCURRENCY CONTROL ALGORITHMS

6.1. Ordering by Validation of Reads Before Writes

The ordering methods proposed below are based upon the order of arrival of the transactions. In certain systems which include the commit feature [LEBI80], the transactions really perform their updates of the database solely at the end of the transaction (commit). Therefore it is possible to consider a transaction as consisting of two phases:

- a read and compute phase,
- an actual write on the database phase (commit).

A simple method can then be defined to prevent the production of non-serializable schedules. This method consists in ordering the transactions according to the time at which they terminate the read and compute phase, and to verify that conflicting accesses to granules occur indeed in the order thus established.

In order to describe this algorithm, we shall introduce an action called VALIDATE [ABAS80] which is supposed to be scheduled by a scheduler as the final step of the read and compute phase. During the execution of VALIDATE, a transaction timestamp equal to the global time is associated to the executing transaction. Furthermore the validation is accepted if all the granules read by the transaction are still in the same state as when they were read and will not be modified by transactions which have already been validated. In the opposite case validation is denied and the transaction is restarted. The verifications are performed by associating with every granule a granule timestamp which is equal to that of the transaction having performed the last write. During

execution of an actual write on the database, the write operation is only carried out if the granule timestamp becomes greater after the write than it was before [THOM79]. This so-called Thomas rule guarantees that obsolete writes are not carried out, while validation guarantees that all the read granules are not obsolete at validation time.

The algorithms for read, write and validation are depicted on figure 26.

In the READ and WRITE algorithms, g is the granule that is accessed by transaction T. S(g) is the timestamp of the granule g while ST(g) is the timestamp of granule g when T reads g. TS(T) is the timestamp associated with transaction T during the execution of VALIDATE.

While the correctness of the proposed algorithm can be demonstrated formally [ABAS80], we shall briefly justify it as follows. The algorithm guarantees that

(1) The actual write operations on the database are carried out in the order of validation of the transactions because of the Thomas rule; therefore the WRITE/WRITE induced precedences respect the order of validation of the transactions.

(2) A transaction can only read the write of another transaction if the latter has been validated while the former has not; this according to the definition of VALIDATE. Therefore the WRITE/WRITE induced precedences also respect the order of validation of the transactions.

(3) A successfully validated transaction cannot have read a granule later written upon by a transaction which was validated before the

former according to the definition of validation. Therefore the READ/WRITE induced precedences also respect the order of validation of the transactions.

Thus finally, all nonpermutable operations are executed upon the same granule in the order of validation of the transactions. This quarantees that there is no circuit in the precedence graph and therefore that the algorithm is correct.

In conclusion, it is worth noting that the ordering by validation algorithm is particularly well suited when each site maintains a differential file [SEVE76], i.e. a special file where updates are stored temporarily and then transferred to the database when the transaction is committed. The validation algorithm must then be executed at each site at the end of a transaction in order to integrate the updates performed by the transaction on the database thus it is executed simultaneously with the first step of commitment when the schedulers verify that all subtransactions have been correctly carried out [LAMS78,ABAS80]. In case of validation failure the transaction is restarted.

6.2. Detection of Circuits in Precedence Graphs

Several concurrency control algorithms allowing a significant degree of parallelism may be constructed so as to detect circuits in transaction precedence graphs and to break them by restarting transactions. However the cost of such algorithms is generally prohibitive especially in a distributed system where the graph may be distributed over several sites.

7. CONCLUSION

It appears that the various methods for concurrency control considered in this paper may be divided into two general classes: (1) timestamp ordering, and (2) locking. In the former case the transactions are ordered either at launching time or at commitment time; thus the ordering may be controlled by the user. In the latter case, it is the requirement of maximum locking before unlocking which defines the ordering; it is thus imposed by the system and cannot be influenced by the user.

It will be recalled that performance requirements have motivated much of the recent work on concurrency control. Therefore it is fitting that the various methods presented here be compared in this regard; unfortunately scant information is available. We may consider this point with respect to degree of parallelism, and time and space overhead. As far as parallelism is concerned, the various algorithms discussed here seem generally similar for indeed optimization through parallel operations has been in all cases the main drive as all algorithms considered seek serializability. Further study is clearly in order. Time cost overhead considerations must include message transmission times and CPU time. The timestamp method requires a timestamp appendage with every read/write message, while locking requires a similar locking appendage. Thus the two methods seem equivalent although timestamps are generally longer than locking requests. Space overhead considerations must take into account the presence of timestamp tables and locking tables which appear generally equivalent. Thus it appears difficult at this time to give preference to any one approach over the

other on the basis of expected performance. Clearly more work is needed both of an analytic and an experimental nature. Finally we should mention that the timestamp method is better in so far as user interface is concerned for the user need not provide any additional information whereas he must specify the locking mode if any.

In summary we believe that the basic principles of concurrency control are fairly well understood. Additional concurrency control algorithms can easily be constructed by varying the time at which ordering is defined, by delaying certain types of actions or by combining the various methods described here. What is most lacking at this time is the availability of reliable performance evaluation based on detailed theoretical analyses and extensive experimentation.

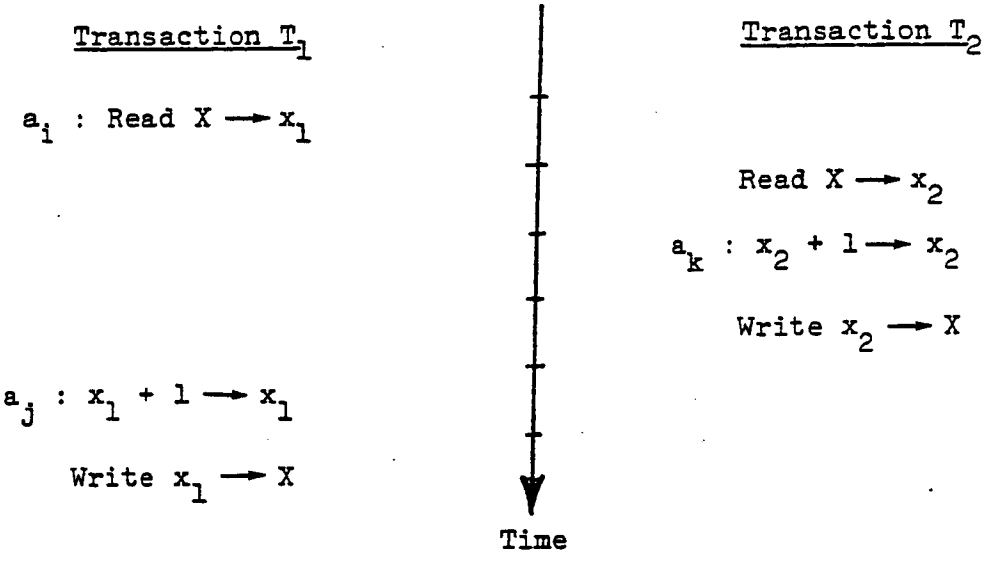

Figure 1. Example of lost operation.

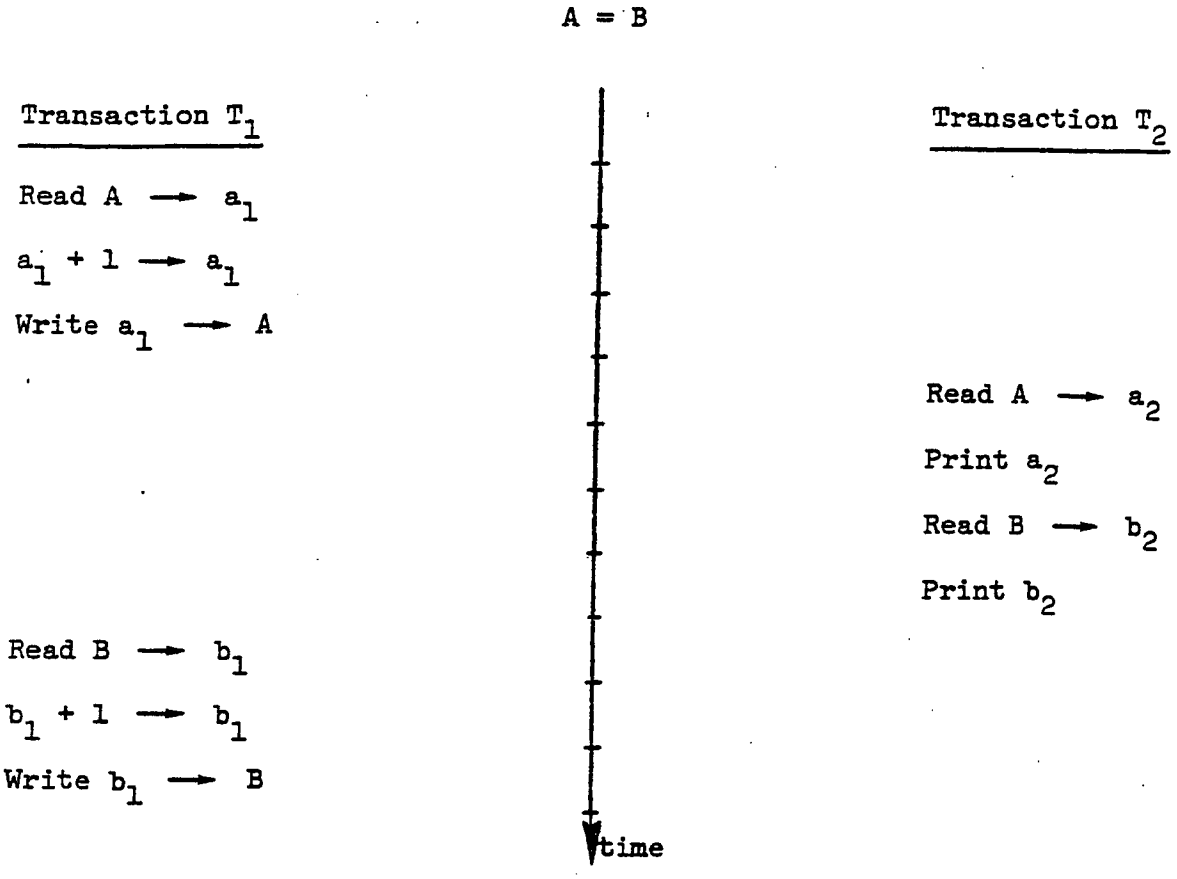

Figure 2. Example of inconsistent output

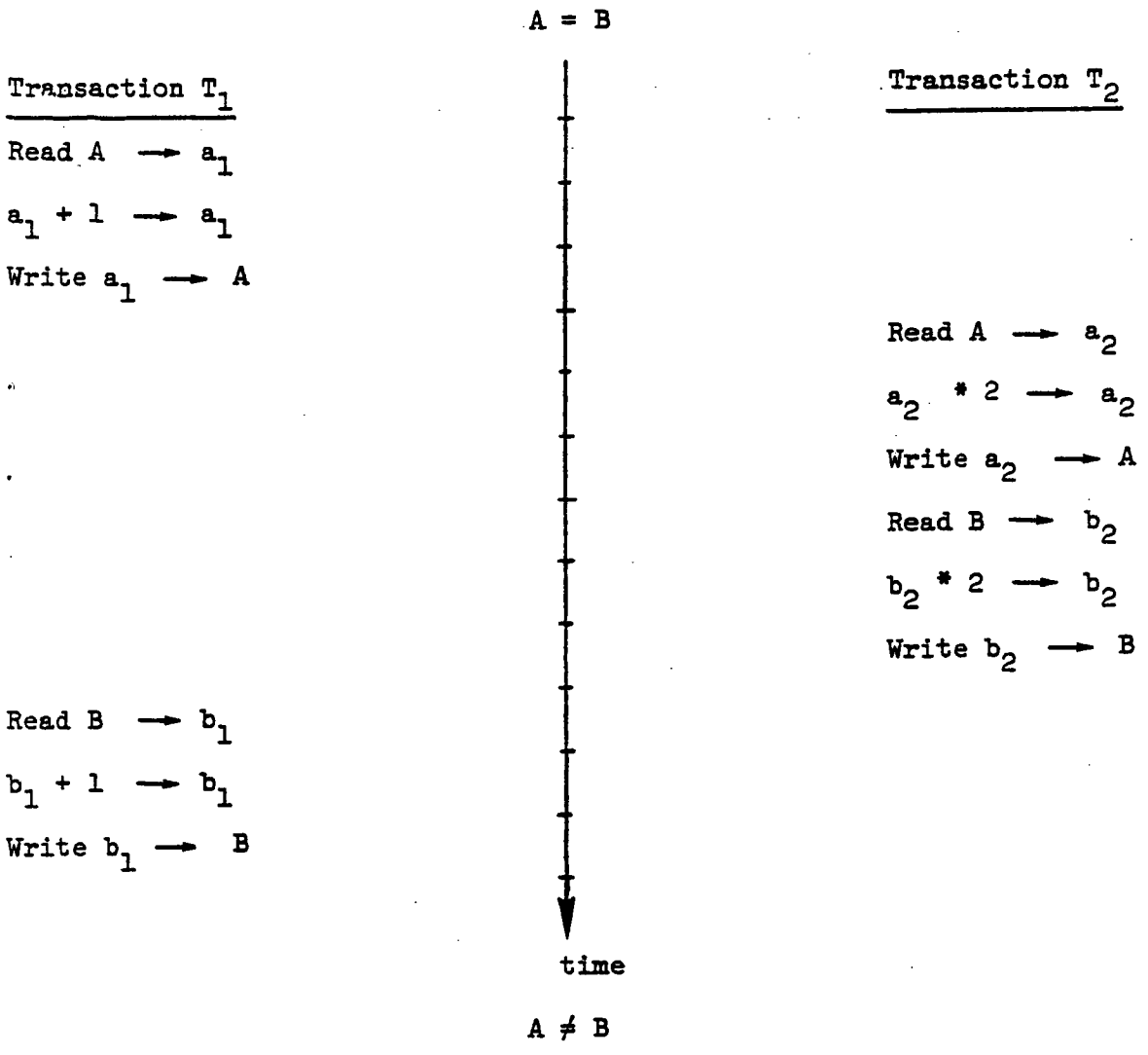

Figure 3. Example of the introduction of inconsistencies in the database

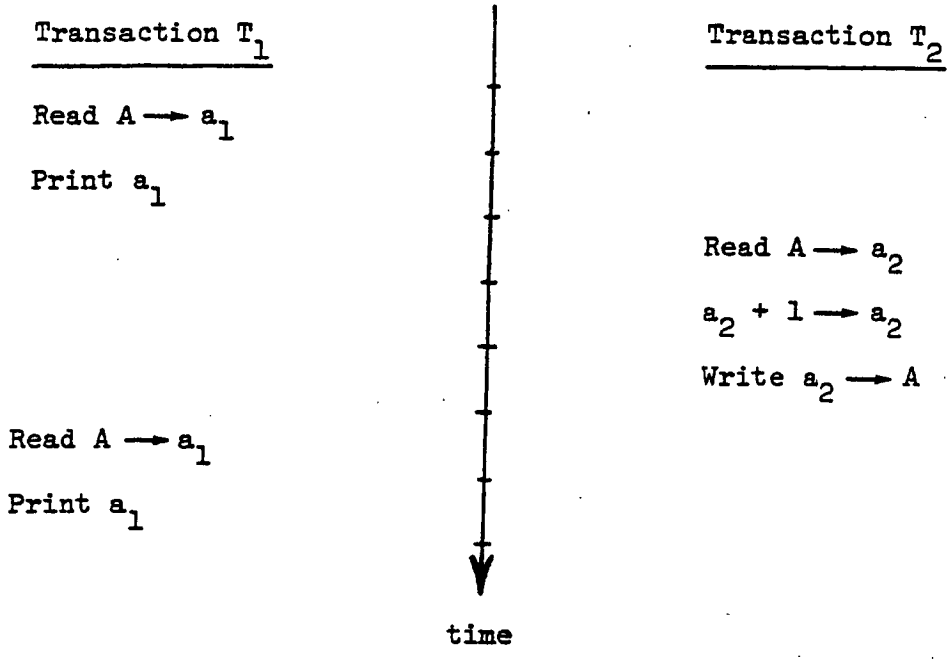

Figure 4. Non repeatable read operations.

Transaction T_1	Transaction T_2
Read A $\rightarrow a_1$	Read A $\rightarrow a_2$
$a_1 + 1 \rightarrow a_1$	$a_2 * 2 \rightarrow a_2$
Write $a_1 \rightarrow$ A	Write $a_2 \rightarrow$ A
Read B $\rightarrow b_1$	Read B $\rightarrow b_2$
$b_1 + 1 \rightarrow b_1$	$b_2 * 2 \rightarrow b_2$
Write $b_1 \rightarrow$ B	Write $b_2 \rightarrow$ B

Figure 5. Transactions T_1 and T_2

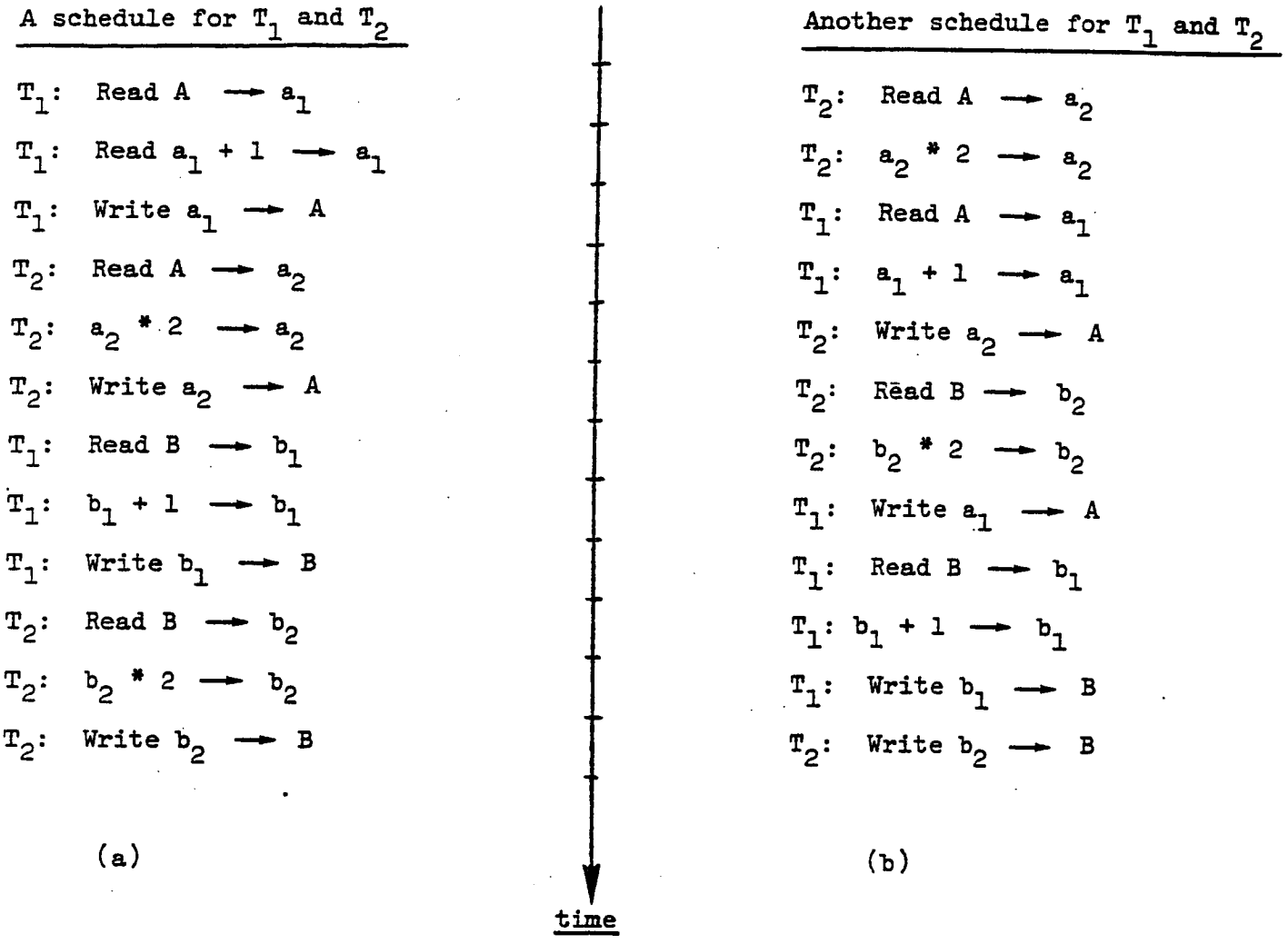

Figure 6. Two schedules for transactions T_1 and T_2 shown in figure 5. It will be seen that schedule (a) is acceptable while schedule (b) is unacceptable as it leads to operation losses.

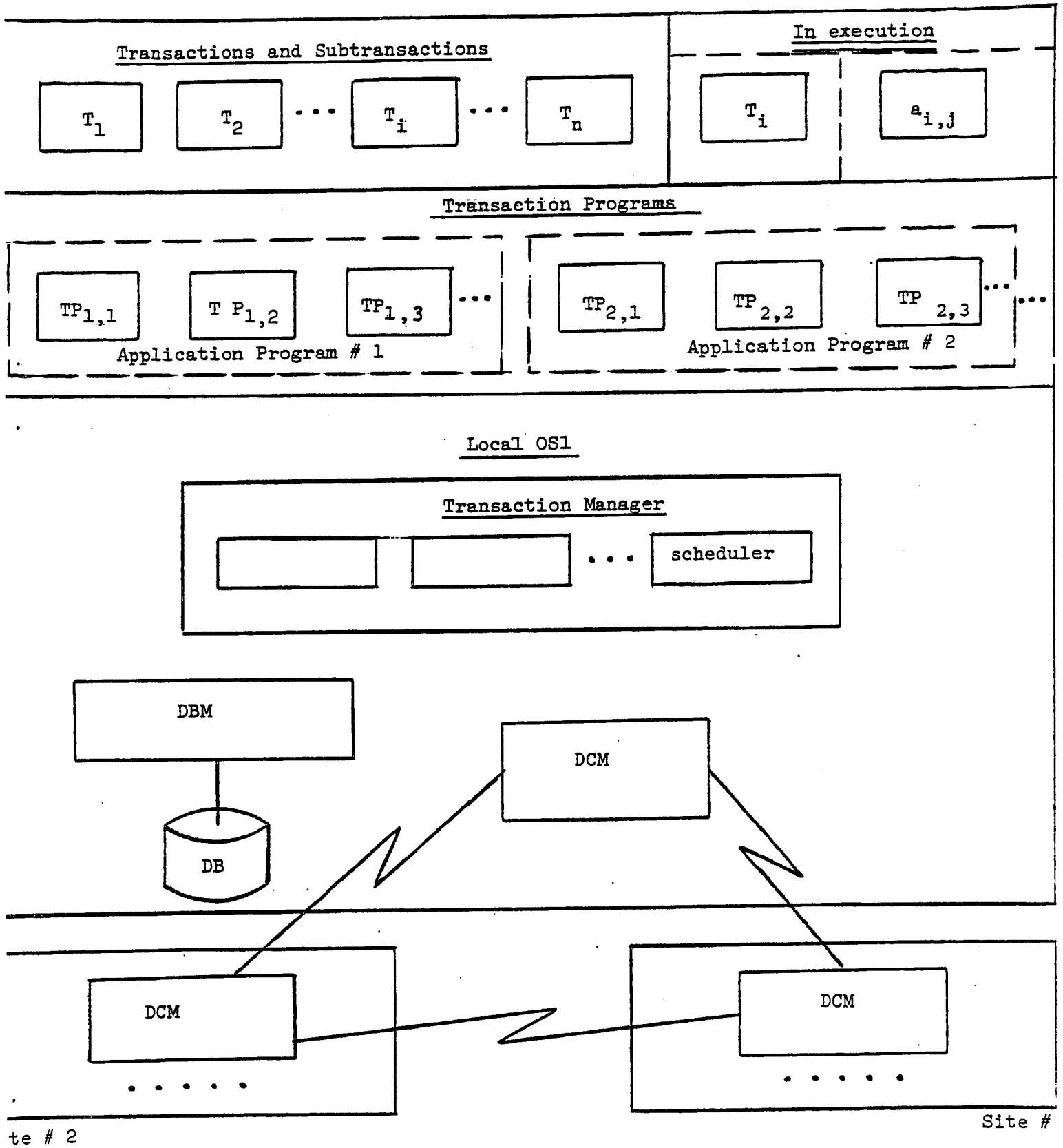

Figure 7. Overview of Execution of Distributed Transaction
$T_1, T_2, \ldots, Tn$: Transactions and substransactions
T_i : Transaction or substransaction currently being processed
$a_{i,j}$: Action j of transaction T_i currently being processed
All three sites have the same internal structures as site #1

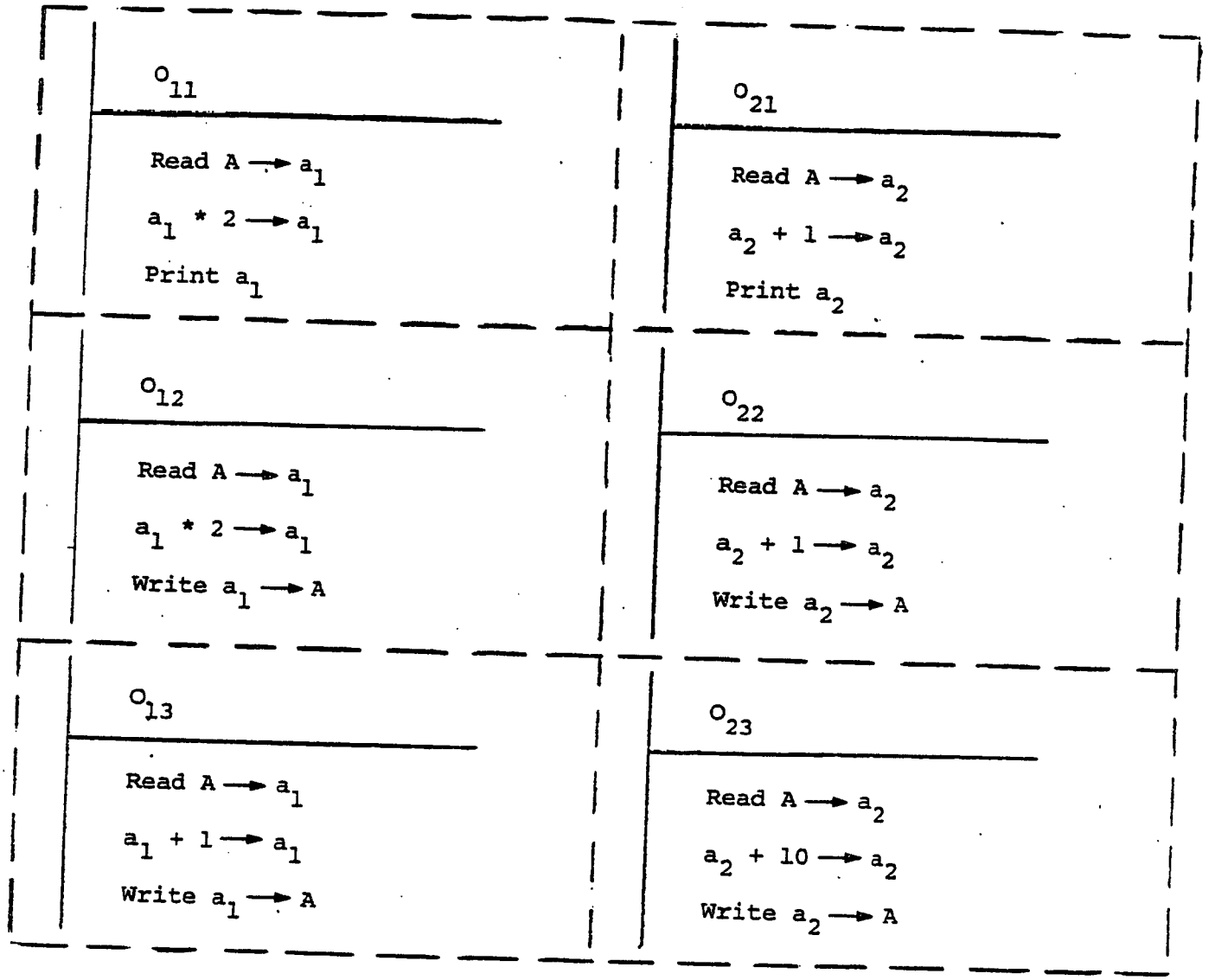

Figure 8: Examples of Operations

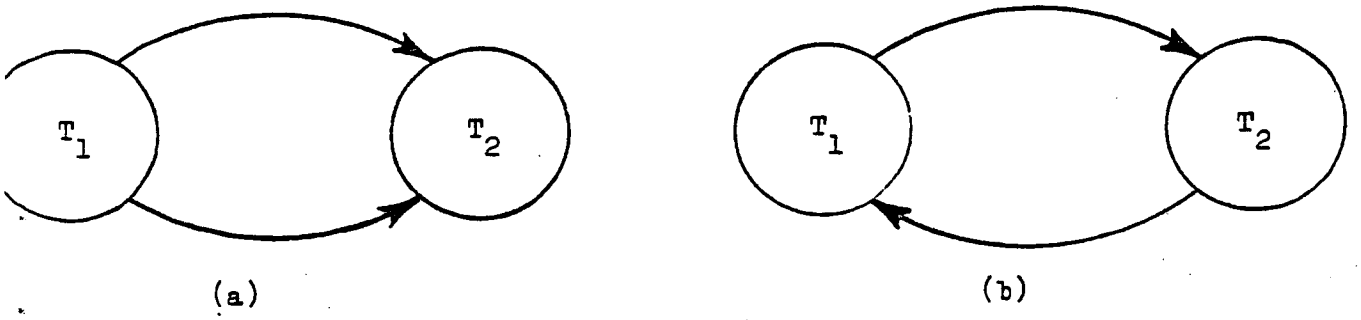

Figure 9. Precedence graphs corresponding to Figure 6.

```
Procedure READ (T_i,g);
     if (S(g) ≤ i)
     then "carry out the read";
          S(g) := i;
     else ABORT;
end READ


Procedure WRITE (T_i,g);
     if (S(g) ≤ i)
     then "carry out the write";
          S(g) := i;
     else ABORT;
end WRITE
```

Figure 10. Total ordering algorithm.

The statements in quotes indicate the presence of statements whose formats depend upon the system.

```
Procedure READ (T_i,g);
     if SW(g) ≤ i
     then "carry out the read";
          SR(g) := MAX(SR(g),i);
     else ABORT;
end READ


Procedure WRITE (T_i,g);
     if (SW(g) ≤ i) ∧ (SR(g) ≤ i)
     then "carry out the write";
          SW(g) := i;
     else ABORT;
end WRITE
```

Figure 11. Partial ordering algorithm.

```
Procedure READ (T_i, g);
     j := "number of the last version of g";
     while SW_j(g) > i
          do j := j - 1;
     "carry out the read of g version j";
     SR_j(g) := MAX(SR_j(g),i);
end READ


Procedure WRITE (T_i, g);
     j := "number of the last version of g";
     while SW_j(g) > i
          do j := j - 1;
     if SR_j(g) > i
     then ABORT;
     else "execute the write inserting a version (j+1) of g"
     SW_j+1(g) := i;
end WRITE
```

Figure 12. Multiversions partial ordering algorithm

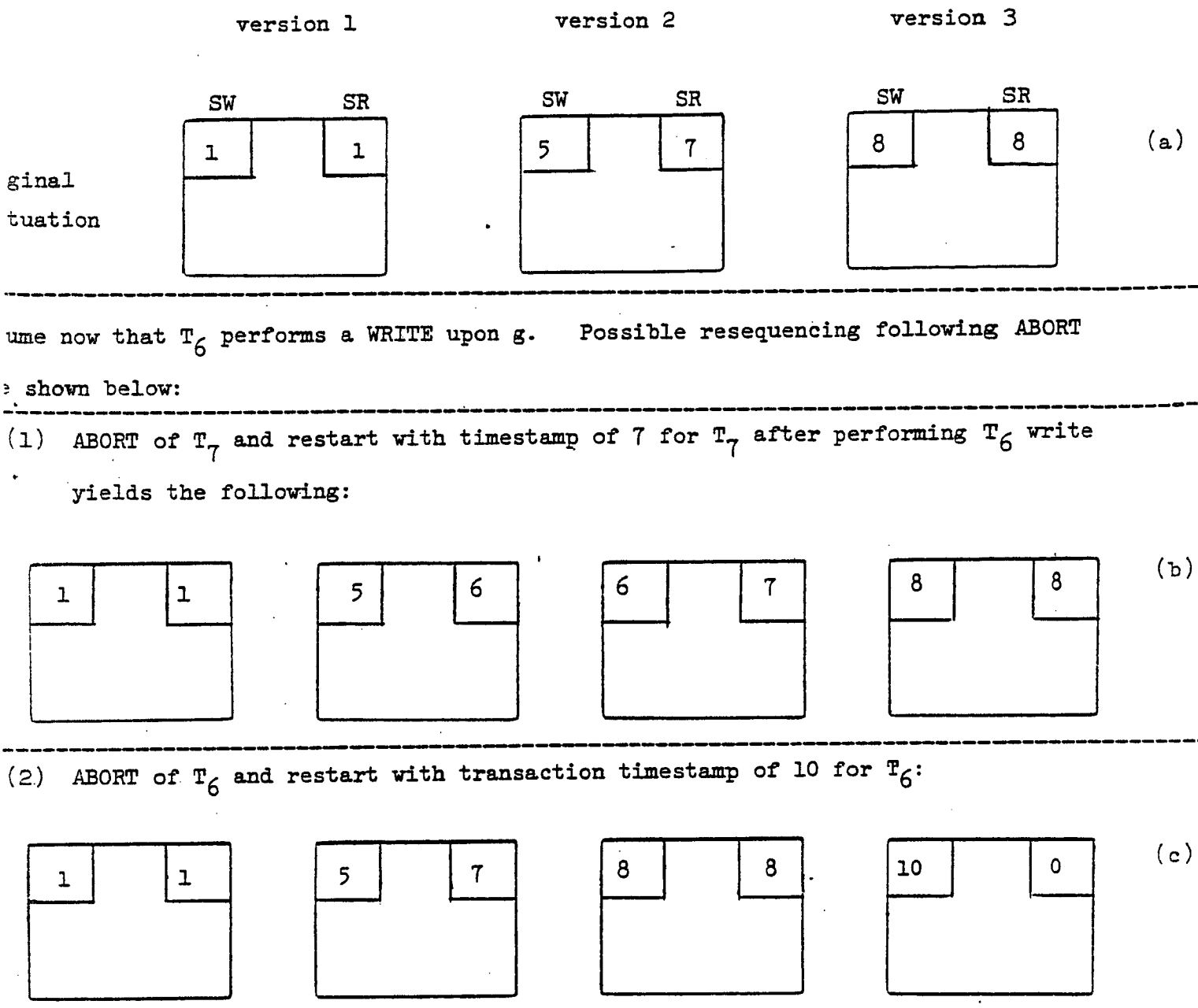

Figure 13. An illustration of some possible sequences of operations controlled by the multiversion partial ordering algorithm.

```
Procedure LOCK (g, M);
    if M ⊂ ¬(¬C * ⋁_{i≠p} A(g, i))
    then A(g, p) := A(g, p) ∨ M;
    else "insert (p, M) into Q[g]";
         "block transaction T_p";
end LOCK

Procedure UNLOCK (g)
    A(g, p) := 0;
    for each (q, M') of Q[g] do
        if M' ⊂ ¬(¬C * ⋁_{i≠p} A(g, i))
        then A(g,q) := A(g, q) ∨ M';
             "remove (q, M') from Q[g]";
             "unblock transaction T_q";
end UNLOCK
```

Figure 14. Locking Algorithm

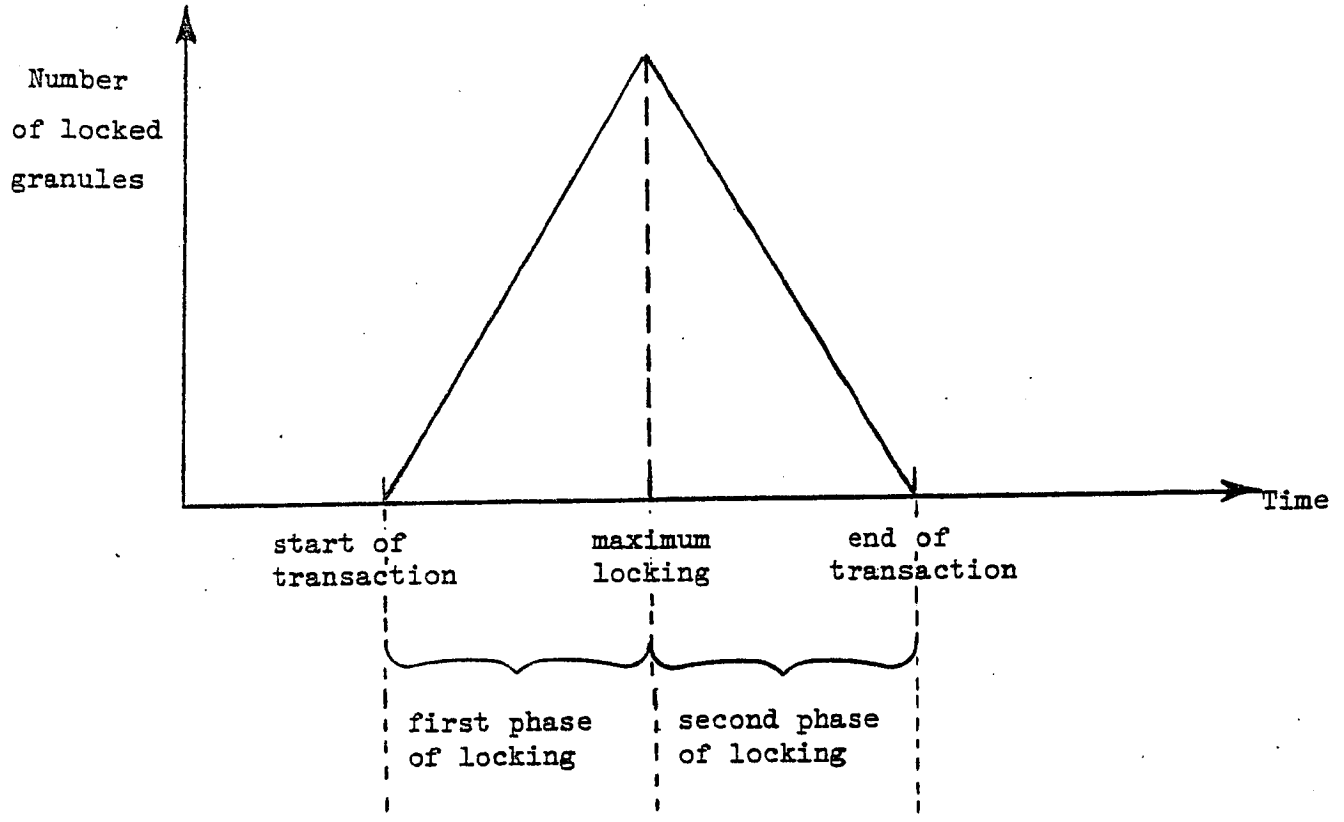

Figure 15. Variation in the number of granules locked by a two-phase transaction

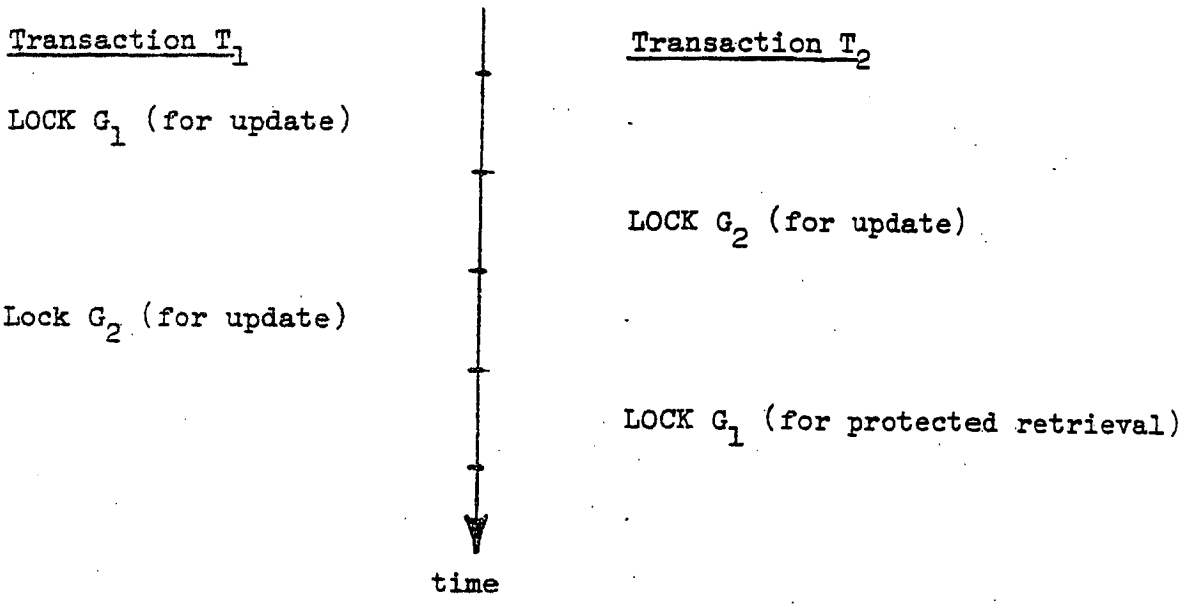

Figure 16. Example of deadlock

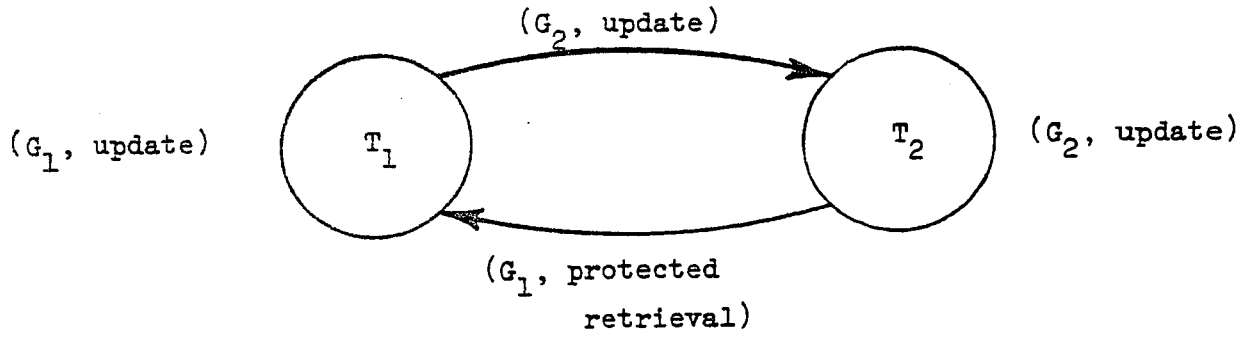

Figure 17. A waiting graph with circuit corresponding to the example shown in figure 16.

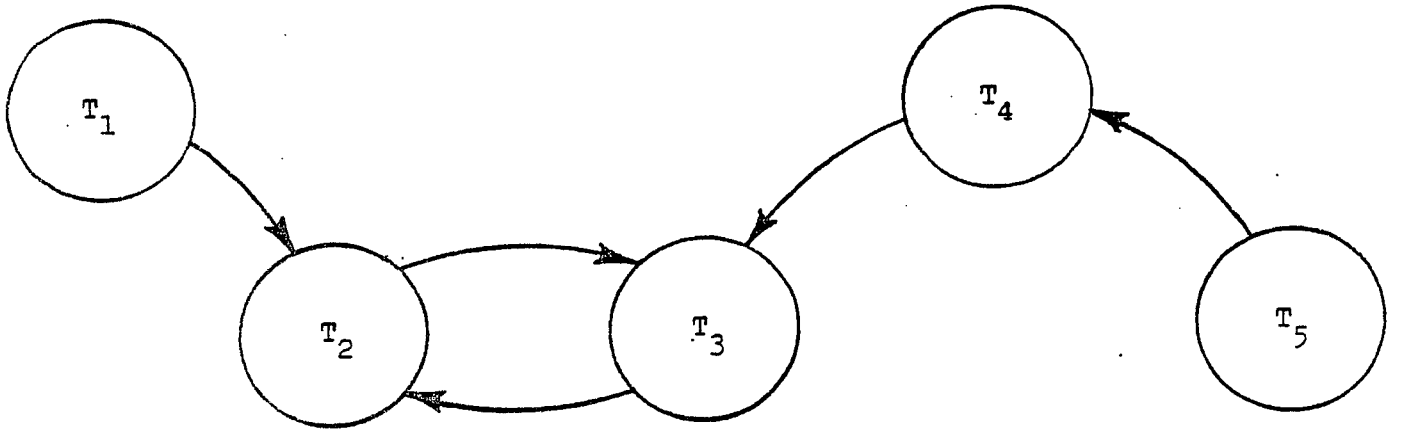

Figure 18. Transactions in various states of deadlock

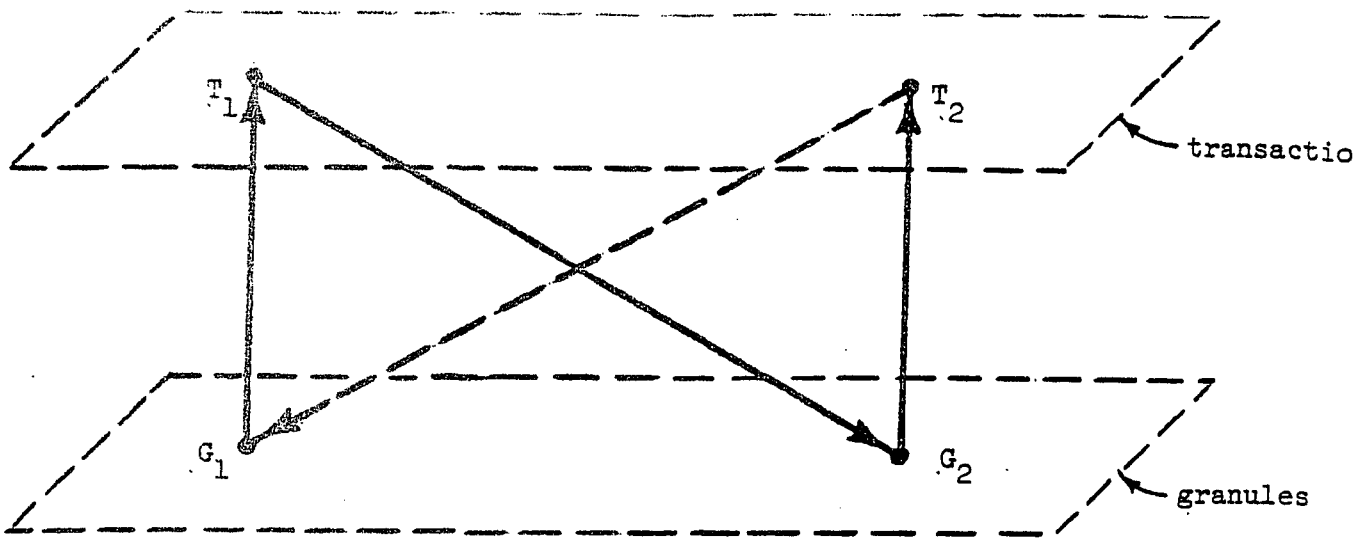

Figure 19. An example of allocation graph corresponding to the example shown in figure 17.

update

protected retrieval

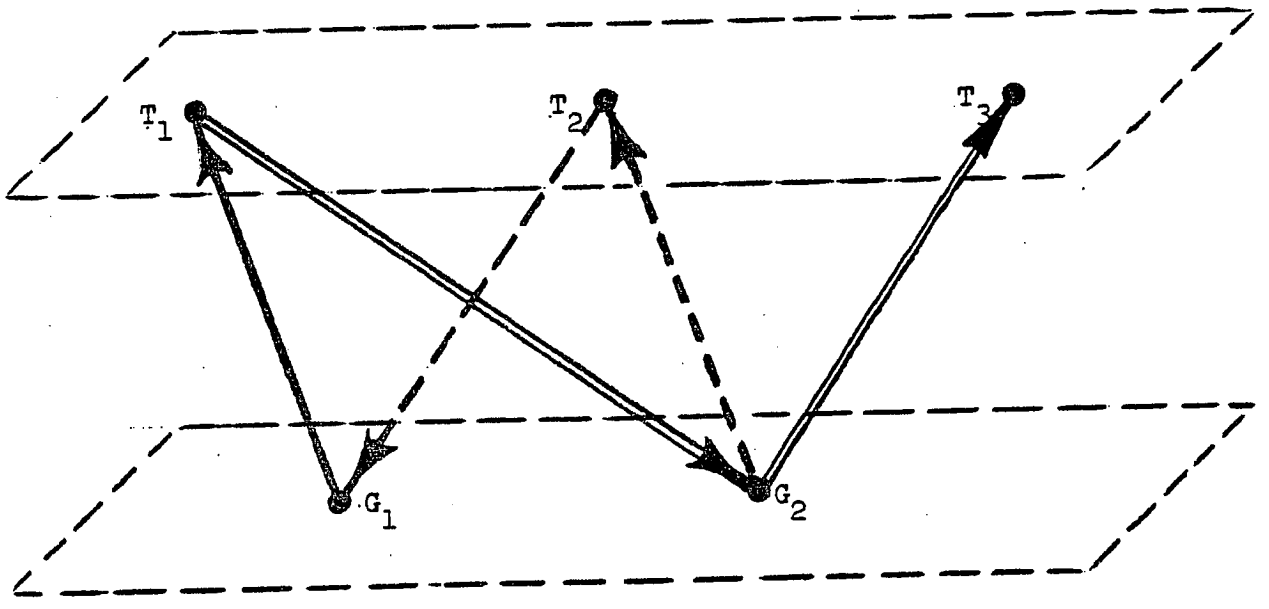

Figure 20. An allocation graph with circuit without deadlock

insertion

update

retrieval

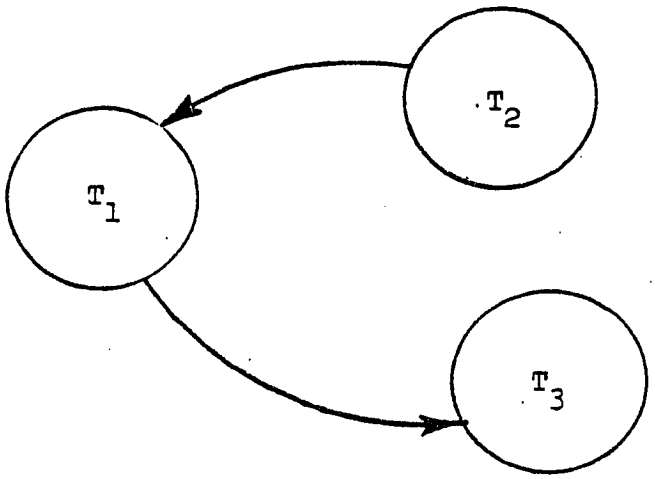

Figure 21. The waiting graph corresponding to the allocation graph of figure 20.

```
Boolean procedure DETECT
begin
    T := { set of transactions T_j such that N(j) ≠ 0 };
    L := { set of granules locked by the transactions ∈ T };
    for each G ∈ L do
      for each "nonmarked request R of T_k waiting for G_i" do
        if SLOCK (G_i, R, k) = true
        then begin
                "mark R";
                N(k) := N(k) - 1;
                if N(k) = φ
                then "remove T_k from T";
                    "add the granules locked by T_k to the set L";
             end
    if T = φ
    then DETECT := false
    else DETECT := true
end DETECT
```

Figure 22. Deadlock detection algorithm

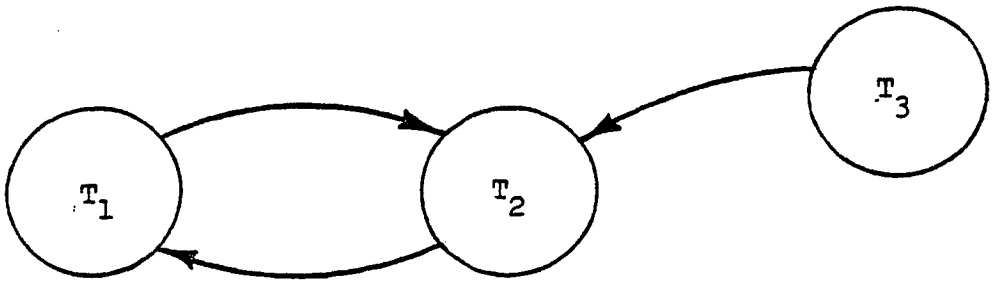

Figure 23. Waiting graph for three deadlocked transactions

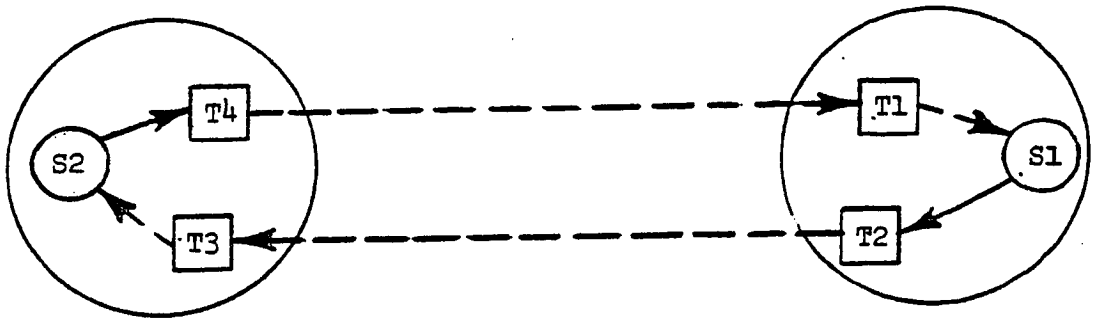

Figure 24. Distributed transaction deadlock

——— allocations

— — — waits

PASSENGERS	
Name	Seat #
JAMES	10
MIKE	5
GEORGE	7
TOPPER	13

FLIGHT INFORMATION	
Flight #	Number of passengers
100	4

Figure 25. An illustration of the phantom problem. Topper is a phantom.

```
Procedure READ (T, g);
     "execute the read";
     ST(g) := S(g);
end READ


Procedure VALIDATE (T);
     for each "granule read by T" do
          if ST(g) ≠ S(g)
          then ABORT(T);
          if "there exists a validated transaction requesting to read g"
          then ABORT(T);
          TS(T) := "global time";
end VALIDATE


Procedure WRITE(T, g);
     if TS(T) ≥ S(g)
     then "perform the write";
     S(g) := TS(T);
end WRITE
```

Figure 26. Ordering by validation algorithm

REFERENCES

[ABAS80]

J. ABASOLO PRIETO

"Processeur de Gestion des Acces Simultanes a une base de donnees repartie"

These de 3e cycle, Paris IX, Juin 1980.

[ADIB78]

M. ADIBA, J.C. CHUPIN, R. DEMOLOMBE, G. GARDARIN, and J. LE BIHAN

"Issues in Distributed Data Base Management Systems: A Technical Overview"

Fourth International Conference On Very Large Data Bases

Berlin, September 1978, p.89-100.

[ALSB76]

P.A. ALSBERG and J.D. DAY

"A Principle for Resilient Sharing of Distributed Resources"

Second International Conference on Software Engineering

New York, 1976, p.562-570.

[BADA80]

D.Z. BADAL

"The Analysis of The Effects of Concurrency Control on Distributed Data Base System Performance"

Sixth International Conference on Very Large Data Bases

Montreal, October 1980.

[BENC76]

E. BENCI, A. CABANES, F. BODART and H. BOGAERT

"Concepts for the Design of a Conceptual Schema"

IFIP Conference on Modelling in DBMS

Freudenstadt, January 1976, North-Holland ed., p.181-200.

[BERN80]

P.A. BERNSTEIN and N. GOODMAN

"Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems"

Sixth International Conference on Very Large Data Bases

Montreal, October 1980.

[BESR80]

P.A. BERNSTEIN, D.W. SHIPMAN and J.B. ROTHNIE

"Concurrency Control in a System for Distributed Data Bases (SDD-1)"

ACM Transactions on Database Systems

Vol. 5, No. 1, March 1980, p.18-51.

[CHU69]

W.W. CHU

"Optimal File Allocation in a Multiple Computer System"

IEEE Transactions on Computers

October 1969, p.885-889.

[CODA71]

Committee on Data Systems Languages

"CODASYL Data Base Task Group Report"

Available from ACM, New York, New York 1971

[CODD70]

E. F. CODD

"A Relational Model of Data for Large Shared Data Banks"

Communications of the ACM

Vol. 13, No. 6, June 1970, p.337-387.

[COFF71]

E. G. COFFMAN, M. J. ELPHICK and A. SHOSHANI

"System Deadlocks"

Computing Surveys Vol. 3, No. 2, 1971, p.67-78.

[COUR71]

P.J. COURTOIS, F. HEYMANS and L. PARNAS

"Concurrent Control with Readers and Writers"

Communications of the ACM

Vol. 14, No. 10, October 1971, p.667-668.

[DATE77]

C.J. DATE

An Introduction to Database Systems

Addison-Wesley Publishing Company

2nd Edition, 1977.

[DIJK68]

E. W. DIJKSTRA

"Cooperating Sequential Processes"

in Programming Languages

Ed. F. Genuys, Academic Press, New York, 1968.

[ENGL76]

R.W. ENGLES

"Currency and Concurrency in the COBOL Data Base Facility"

IFIP Conference on Modeling in DBMS

Freudenstadt, January 1976, North Holland ed, p.339-363.

[ELLI77]

C.A. ELLIS

"A Robust Algorithm for Updating Duplicate Databases"

Second Berkeley Workshop on Distributed Data Management and Computer Networks

Berkeley, May 1977, p.146-158.

[ESWA76]

K.P. ESWARAN, J.N. GRAY, R.A. LORIE and L.L. TRAIGER

"The Notion of Consistency and Predicate Locks in a Database System"

Communications of the ACM

Vol. 19, No. 11, November 1976, p.624-633.

[FAGI77]

R. FAGIN

"Multivalued Dependencies and a New Normal Form for Relational Databases"

ACM Transactions on Database Systems

Vol. 2, No. 3, September 1977, p.262-278.

[GARD76]

G. GARDARIN

"Integrity of Databases: A General Locking Algorithm with Deadlock

Avoidance"

IFIP Conference on Modelling in DBMS

Freudenstadt, January 1976, North-Holland ed., p.395-411.

[GARD77]

G. GARDARIN and P. LEBEUX

"Scheduling Algorithms for Avoiding Inconsistency in Large Data Bases"

Third International Conference on Very Large Data Bases

Tokyo, October 1977, p.501-506.

[GARD78]

G. GARDARIN

"Resolution des Conflits d'Acces Simultanes a un Ensemble d'Informations-Applications aux Bases de Donnees Reparties"

These d'Etat, Paris VI, April 1978.

[GARD79]

G. GARDARIN and M. MELKANOFF

"Proving Consistency of Database Transactions"

Fifth International Conference on Very Large Data Bases

Rio De Janeiro, October 1979, p.291-298.

[GARD80]

G. GARDARIN

"Integrity, Consistency, Concurrency and Reliability in Distributed Data Base Management Systems"

Conference on Distributed Data Bases

Paris, 1980, North-Holland Ed., p.335-351.

[GOLD77]

B. GOLDMAN

"Deadlock Detection in Computer Networks"

MIT Thesis, Boston, June 1977.

[GRAY78]

J. GRAY

"Notes on Data Base Operating Systems"

IBM Research Report RJ 2188 (30001)

San Jose, February 1978.

[HOLT72]

R.C. HOLT

"Some Deadlock Properties of Computer Systems"

Computing Surveys

Vol. 4, No. 3, September 1972, p.179-196.

[HOWA73]

J.H. HOWARD

"Mixed Solutions for the Deadlock Problem"

Communications of the ACM

Vol. 16, No. 7, July 1973, p.427-430.

[LAMP78]

L. LAMPORT

"Time, Clocks and the Ordering of Events in a Distributed System"

Communications of the ACM

Vol. 21, No. 7, July 1978, p.558-565.

[LAMS78]

B.LAMPSON AND H. STURGIS

"Crash Recovery in a Distributed Data Storage System"

Internal Report, Computer Science Laboratory

Xerox Palo Alto Research Center, 1976.

[LEBI80]

J. LE BIHAN, et al.

"SIRIUS: A French Nationwide Project on Distributed Data Bases"

6th International Conference on Very Large Data Bases

Montreal, 1980.

[LELA78]

G. LE LANN

"Algorithm for Distributed Data Sharing which Use Tickets"

Third Berkeley Workshop on Distributed Data Management and Computer Network

Berkeley, August 1978, p.259-272.

[LIND79]

B. G. LINDSAY et al.

"Notes on Distributed Databases"

IBM Research Report RJ 2571 (33471)

San Jose, July 1979.

[MENA78]

D. A. MENASCE

Ph.D. Dissertation

"Coordination in Distributed Systems: Concurrency, Crash Recovery and Data Base Synchronization" UCLA, December 1978.

[MENA80]

D. A. MENASCE, G. J. POPEK and R. R. MUNTZ

"A Locking Protocol for Resource Coordination in Distributed Databases"

ACM Transactions on Database Systems Vol. 5, No. 2, June 1980.

[MURP68]

J. E. MURPHY

"Resource Allocation with Interlock Detection in a Multi-Task System"

AFIPS-FJCC

Vol. 33/P2, 1968, p.1169-1176.

[PAPA79]

C. H. PAPADIMITRIOU

"Serializability of Concurrent Updates"

J. of the ACM

Vol. 26, No. 4, October 1979, p.631-653.

[PARE77]

C. PARENT

"Integrity in Distributed Data Bases"

AICA 1977 Pisa, October 1977, Vol. II, p.199-212.

[RIES77]

D.D. RIES AND M.R. STONEBRAKER

"Effects of Locking Granularity in a Database Management System"

ACM Transactions on Database Systems

Vol. 2, No. 3, September 1977, p.233-246.

[RIES79]

D.D. RIES and M.R. STONEBRAKER

"Locking Granularity Revisited"

ACM Transactions on Database Systems

Vol. 4, No. 2, June 1979, p.210-227.

[ROSE78]

D.J. ROSENCRANTZ, R.E. STEARNS and P.W. LEWIS

"System Level Concurrency Control for Distributed Database Systems"

ACM Transactions on Database Systems

Vol. 2, No. 3, September 1977, p.233-246.

[SCHL79]

G. SCHLAGETER

"Enhancement of Concurrency in Database Systems by the Use of Special Rollback Methods"

IFIP Conference on Data Base Architecture

Venice, June 1979, North Holland Ed., p.141-150.

[SEVE76]

D. G. SEVERANCE and G. M. LOHMAN

"Differented Files: Their Applications to the Maintenance of Large Database Systems"

ACM Transactions on Database Systems

Vol. 1, No. 3, September 1976, p.256-367.

[STON78]

M. R. STONEBRAKER

"Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES"

Third Berkeley Workshop on Distributed Data Management and Computer Network
Berkeley, August 1978, p.235-258.

[THOM79]
R. H. THOMAS
"A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases"
ACM Transactions on Database Systems
Vol. 4, No. 2, June 1979, p.180-209.

[TRAI79]
I. L. TRAIGER, J. N. GRAY, C. A. GALTIERI and B. G. LINDSAY
"Transactions and Consistency in Distributed Database Systems"

[TRIN75]
M. TRINCHIERI
"On Managing Interference Caused by Database Sharing"
Alta Frequenza, Vol. XLIV, No. 11, 1975, p.641-650.

[ULLM80]
J.D. ULLMAN
Principles of Database Systems
Computer Science Press, Potomac, Md., 1980

[ZANI76]
C. ZANIOLO
"Analysis and Design of Relational Schemata for Database Systems"
UCLA Computer Science Department Report
UCLA - ENG - 7669, July 1976.