

590

A Study of Synchronization Techniques
in
Distributed Database Systems

Dissertation submitted in partial fulfilment of
the requirements for the Degree of
MASTER OF PHILOSOPHY

PARTHA SARATHI ACHARYA


121p-

**SCHOOL OF COMPUTER AND SYSTEM SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI—110067
1985**

CERTIFICATE

This work embodied in this dissertation has been carried out at the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi-110067. This work is original and has not been submitted so far, in part or full, for any other degree or diploma of any University.


(PARTHA SARATHI ACHARYA)
Student


(PROF. K.K. NAMBIAR)
Dean


(DR. P.C. SAXENA)
Supervisor

School of Computer and Systems Sciences
Jawaharlal Nehru University
New Delhi - 110067

ACKNOWLEDGEMENTS

I wish to express my deep sense of gratitude to my Supervisor, Dr. P.C. Saxena, Assistant Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, for his valuable guidance, enthusiastic cooperation and encouragement. He has been a constant source of inspiration throughout this work.

I am grateful to Professor K.K. Nambiar, Dean, School of Computer and Systems Sciences, Jawaharlal Nehru University for his cooperation and providing me with all the facilities in completing this work.

I am thankful to my friend and classmate Mr. R.C. Tripathy for his timely help and discussions.

My thanks are also due to the faculty and other staff for their cooperation and help in many ways.

I also thank Mr. S.K. Sapra for typing the dissertation so neatly.

Finally, I am grateful to Council of Scientific and Industrial Research, New Delhi for providing me with the financial assistance that made the work possible.

PS Acharya
(PAKTHA SARATHI ACHARYA)

Preface

Distributed database system, a recent evolution in database technology has emerged from the successful combination of databases and computer networks. In such a system, an integrated database is built on top of a computer network rather than on a single computer. The data constituting the database are placed at various sites of the computer network, and individual application programmes run by the corresponding computers access and update data at different sites.

The distributed database technology faces completely new problems and a great amount of research work has been done in order to solve them. Concurrency control is one such problem which gets complexified due to the distributed nature of the database system in contrast to the centralized systems where the problem is relatively simpler. The synchronization techniques studied in this dissertation have been designed to solve the problem of concurrency control in distributed systems.

This work has been divided into five chapters. Chapter 1 presents an overview of distributed databases which is an introduction to distributed systems. Chapter 2 discusses the problem of concurrency control that arises in such systems. Chapter 3 contains the basic synchronization

techniques viz. two-phase locking and timestamp ordering, suggested and used to address this problem. Chapter 4 discusses two advanced techniques viz. Conflict analysis and reservation list that eliminate some of the shortcomings encountered in the techniques of the preceding chapter. Chapter 5 embodies the techniques suggested by integrating the basic methods of two-phase locking and timestamp ordering.

At the end, a list of references has been given for further reading.

CONTENTS

	<u>Page</u>
CERTIFICATE	
ACKNOWLEDGEMENTS	
PREFACE	
CHAPTER 1 AN OVERVIEW OF DISTRIBUTED DATABASES	1-19
1.1 Distributed database	2
1.2 Motivations for distributed systems	6
1.3 Comparative features of distributed and centralized databases	8
1.4 Distributed database management system	12
1.5 Distributed database applications	16
CHAPTER 2 CONCURRENCY CONTROL	20-42
2.1 What is concurrency control	20
2.2 Transaction-processing model	26
2.2.1 Centralized transaction processing model	26
2.2.2 Distributed transaction processing model	29
2.3 Transaction and consistency	31
2.4 Serializability	33
2.4.1 Serializability in a centralized database	35
2.4.2 Serializability in a distributed database	39

	Pages
CHAPTER 3	
SYNCHRONIZATION TECHNIQUES BASED ON TWO-PHASE LOCKING AND TIMESTAMP ORDERING	43-78
3.1 Two-phase locking	43
3.1.1 Case of centralized database	43
3.1.2 Case of distributed database	50
3.1.3 Management of distributed deadlocks	56
3.2 Timestamp ordering (T/O) techniques	69
3.2.1 The Basic timestamp mechanism	70
3.2.2 The conservative-timestamp ordering method	75
CHAPTER 4	
SYNCHRONIZATION TECHNIQUES BASED ON CONFLICT GRAPHS AND RESERVATION LISTS	79-97
4.1 Conflict analysis	79
4.1.1 Conflict graphs	81
4.1.2 Timestamp-based protocols	85
4.2 Reservation Lists	86
CHAPTER 5	
INTEGRATED CONCURRENCY CONTROL	98-117
5.1 Decomposition of concept of serializability	99
5.2 Integrated concurrency control methods	102
5.3 Conclusion	115
BIBLIOGRAPHY	118-121

CHAPTER - 1

AN OVERVIEW OF DISTRIBUTED DATABASES

For last fifteen years, computers have been extensively used for building powerful and integrated database systems. Such database systems have found wide-ranging applications in various fields like commercial, scientific, technical and other organizations. However, in recent years availability of low cost computers and of computer networks has given rise to a new type of system which eliminates many of the short-comings of centralized databases and fits more naturally in the decentralized structures of many organizations. This system is known as distributed database system which, unlike the centralized ones, has databases stored with different computers at different sites of a computer network.

This chapter formally introduces a distributed database system. Section 1.1 presents a precise definition of distributed databases followed by the motivations leading to the organization of distributed database system in section 1.2. Section 1.3 presents a comparative picture of various features of distributed and centralized systems. Preliminary ideas and architecture of a distributed database management system (DDBMS) required for the understanding of synchronization techniques have been presented in section 1.4. The last section lists the areas or organizations of distributed database applications.

1.1 Distributed database

A distributed database is a collection of data distributed over different computers of a computer network and it is characterized by the following (CER184) :

- i) Each of the computers (i.e. processor along with its memory and peripheral devices) of the network is referred to as a site which has autonomous processing capability.
- ii) Each site also participates in the execution of global applications or distributed applications in which a site might require to access data residing at more than one site. The existence of global applications is considered the discriminating characteristic of distributed databases with respect to a set of local databases.

Illustration

Suppose there is a bank having more than one branch (say three) situated at geographically different locations. Each branch has a computer with one or more than one teller terminals and the computer controls the account database of that branch. Each computer with its local account database at one branch constitutes one site of the distributed database. Computers at various branches are inter connected by a communication network. Such a system is known as distributed database system (Figure 1).

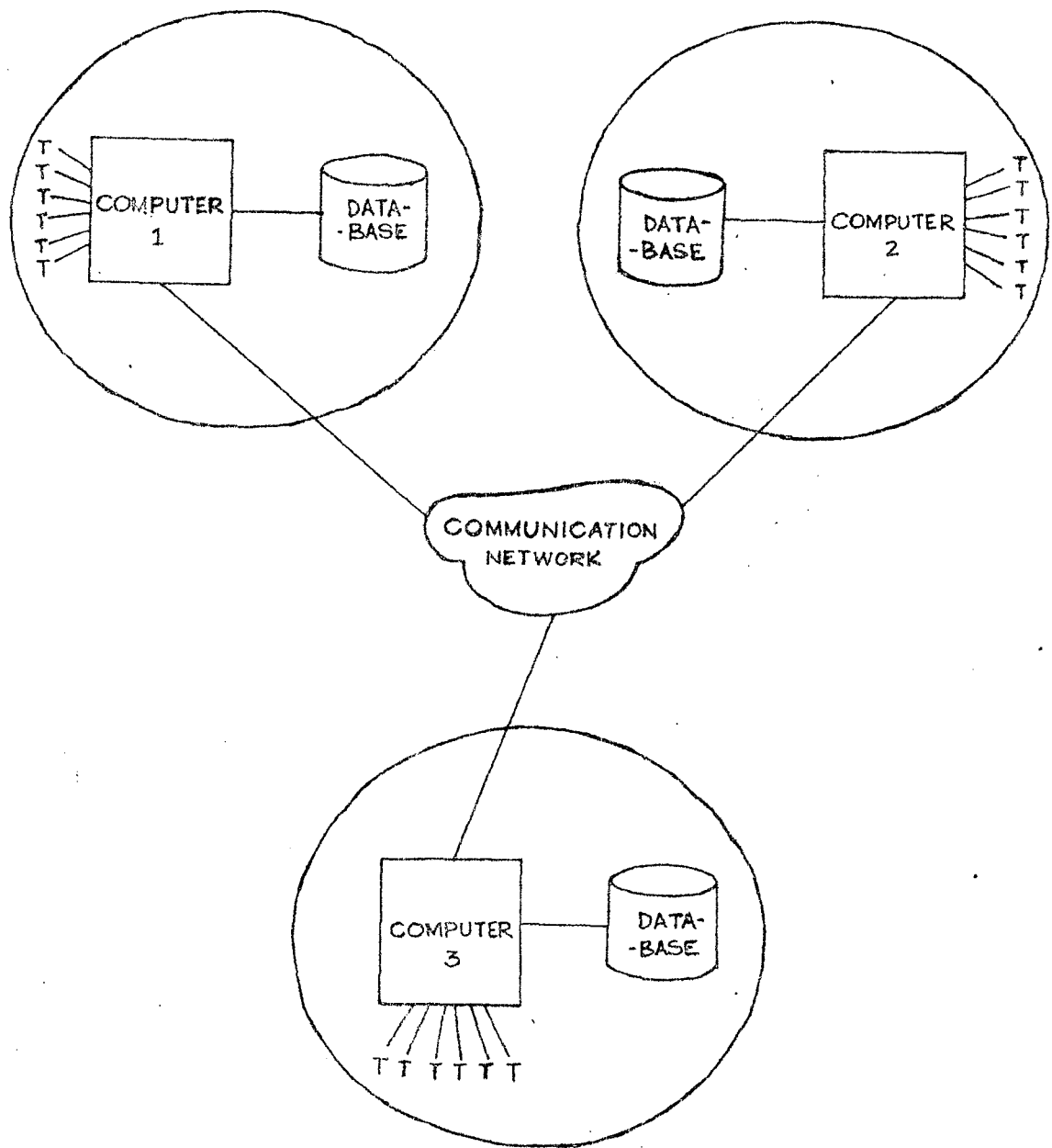


FIG.1 A DISTRIBUTED DATABASE ON A GEOGRAPHICALLY DISPERSED NETWORK.

Applications issued by teller terminals of a particular branch may need to access only the account database of that branch. These applications are independently processed by the computer of that branch and are called local applications. A debit or credit application on an account stored at the same branch at which it is issued is an example of local application.

An application requested to transfer funds from an account of one branch to an account of another branch requires updating the database at two different sites. Such an application is called a global application.

Distributed databases can also be built on local networks unlike the preceding example where the databases are placed at geographically different locations.

Illustration

Suppose the computers and corresponding databases of the above example are removed to a common building and are connected with a local network. Then each processor and its database constitute a site of the local computer network and the system is also known as distributed database system because the characteristics of distributed databases remain satisfied (Figure 2).

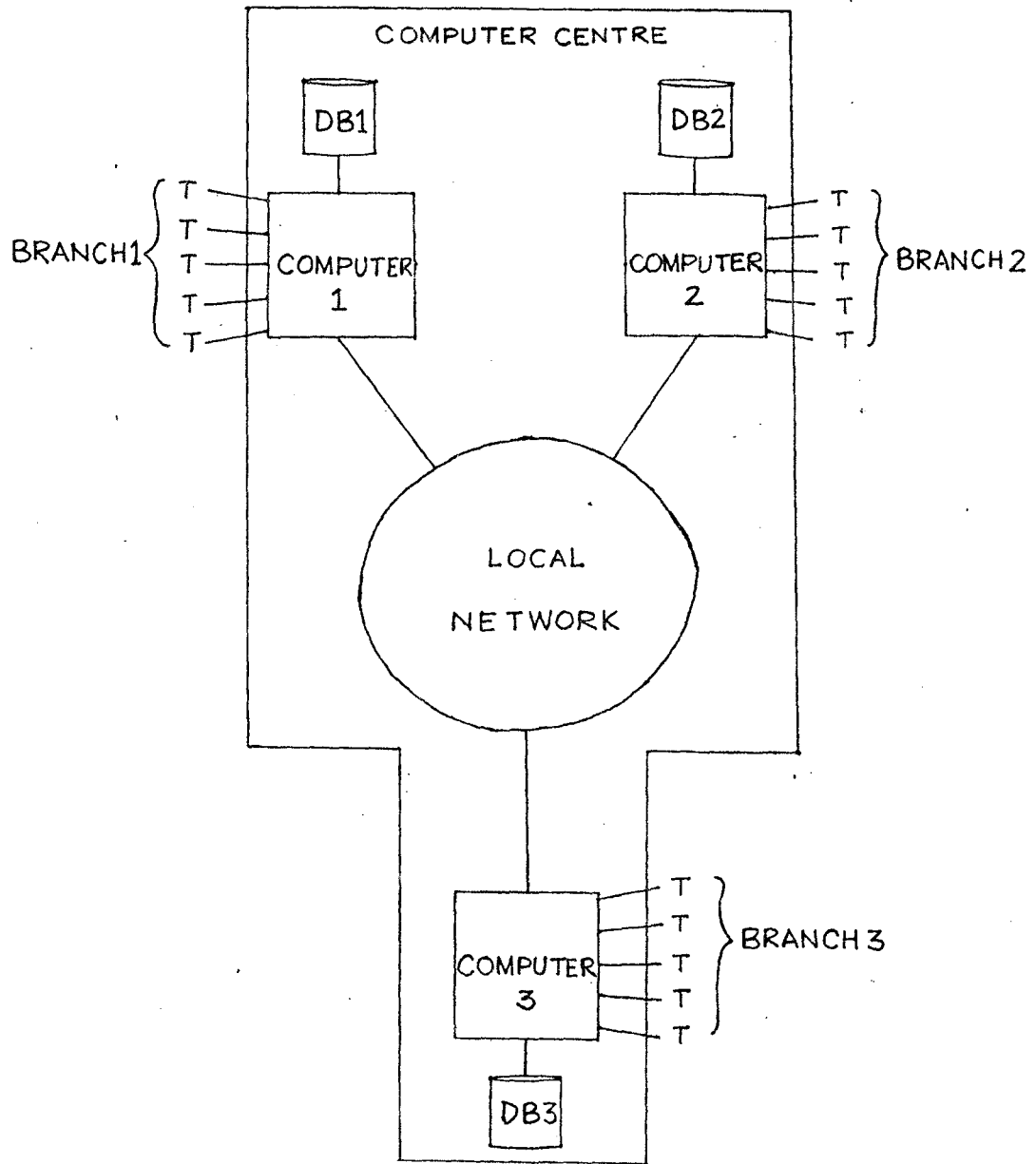


FIG.2 A DISTRIBUTED DATABASE ON A LOCAL NETWORK

1.2 Motivations for distributed systems

Technological changes like price-performance revolutions in micro-electronics, development of efficient communication systems and growing complexity of user needs are the major motivations for distributed database systems (DAVI 81).

Microelectronics technology

Technological advances like low-cost manufacturing of large scale and very large scale integrated circuits (LSI & VLSI) and large-sized memory chips have brought in a falling trend in hardware price. This has made it easier and cheaper to install a multi-computer system (both in centralized as well as distributed database systems) consisting of several processors than to invest in a large and complex multiprogrammed uniprocessor.

Communication technology

Use of simple and cheap technologies as twisted pairs, coaxial cables, micro-wave transmission as well as sophisticated technologies like fibre optics in local area computer networks has prompted the building of distributed database systems out of several processing elements.

User needs

Usually, organizations like industries, banks, inventory systems, hospitals and public administration

systems exhibit decentralized functional structures because activities in these organizations are decentralized by nature. Thus a decentralized style of management is more suitable for such organizations in contrast to conventional centralized style. For instance, it may be more profitable to provide each department of an organization with its own small computer and the required database of that department. Consequently, local tasks of a department are run and controlled by the people of that department who understand them best; in addition, they have other databases of the system placed in various departments at their disposal due to the network system. Thus a distributed database system simplifies the task of decision-making and hence the task of management and improves overall efficiency.

Distributed database system is also economic. The possibility of installing processing elements with required databases (i.e. databases containing informations relevant for particular locations) at various locations brings in the advantage of reduced communication cost. Besides, in a distributed system, a lot of processing can be conducted on local computers in contrast to all processings being handled by one central but remote big mainframe computer.

Distributed database approach supports a smooth incremental growth where an organization grows by adding new, relatively autonomous organizational units (new branches, new warehouses etc.). Such an addition, unlike in case of centralized database, does not affect the functioning of already existing units.

In distributed database systems, database can be replicated at each other site (fully redundant database) or at some of the sites (partially redundant database) depending on the need of the user. Such replications provide the system with higher reliability and availability. Because, failure of a particular site does not prevent the system from being operational. If the system does not contain redundant data, effect of each failure is confined to those applications which use the data of the failed site. Availability and faster access to data is achieved due to the possibility of storing portions of the database near to where they are frequently used.

1.3 Comparative features of distributed & centralized databases

Distributed databases allow design of systems which has different features from traditional centralized systems. Centralized control, data independence,

reduction of redundancy, integrity, recovery concurrency control, privacy and security are the various features that characterize the traditional database approach.

Centralized control

Centralized control is an essential feature of traditional systems because it provides an organization with a central command over its information resources. The database administrator guarantees the safety of the data.

However, in distributed databases centralized control is de-emphasized. A global database administrator takes care of the whole database whereas the local database administrators have the responsibility of their respective local databases, often with a high degree of site autonomy. A distributed database may also be designed with global database administrators accompanied with complete centralized control.

Data independence

Data independence is a major objective of centralized database system and is defined as the immunity of application programmes to the actual organization of data. It has the advantage that programmes are not affected by changes in storage structures and access strategy.

In distributed database, data independence has the same importance as in traditional systems in addition to a new aspect, known as distribution transparency. Distribution transparency provides a centralized view of the databases to the programmes implemented in the system. Consequently, the movement of data from one site to another does not affect the correctness of programmes, though the speed of execution gets affected.

Reduction in data redundancy

In centralized systems, reduction in data redundancy is desired to avoid inconsistency and to prevent wastage in storage space.

However, in case of distributed databases, redundancy is a desirable feature; because replicated copies of databases guarantee reliability of the system and enhances its availability in spite of wastage in storage space. The problem of inconsistency arises when updates are not performed consistently on all copies. This problem is related to concurrency control which has been discussed in chapter 2.

Integrity, recovery and concurrency control

The problems of integrity, recovery and concurrency control in traditional database system are resolved by the use of transactions. A transaction is

a sequence of executable operations which either are performed in entirety or are not performed at all and thus is an atomic unit of execution. For instance, the debit operation is a transaction which is either executed or none is executed.

Same approach is made to these problems in case of distributed database systems where the problems are further complexified due to distribution. Atomic transactions ensure integrity of the database; however, the atomicity is threatened by site failures or concurrent execution of different transactions. Site failures may cause the system to stop in the midst of transaction execution, thereby violating the atomicity requirement. Concurrent execution of two or more transactions may permit one transaction to observe an inconsistent, transient state created by another transaction during its execution.

Recovery and synchronization techniques take care of the problem of preserving the transaction atomicity during site failures and concurrent execution of transactions.

Privacy and security

In traditional databases, privacy and security are ensured by the database administrators having centralized control and specialized control procedures.

In case of distributed databases with a high degree of site autonomy, privacy is maintained by the local database administrators; but the security is threatened because of the communication network.

1.4 Distributed Database Management System

A distributed database management system (DDBMS) is a collection of sites interconnected by a network (DEPP 76, ROTH 77). Each site is a computer with one or both the following software modules : a transaction manager (TM) or a data manager (DM). TMs supervise interactions between users and the DDBMS while DMs interact with the database. All the sites are interconnected by a network which is a computer-to-computer communication system. The network is assumed to be perfectly reliable with the following required conditions : firstly, the communication system is capable of transmitting messages between sites without distortion or error; secondly, between any pair of sites the network delivers messages in the order they are sent.

Database

The database in DDBMS consists of a collection of logical data items, denoted by X, Y, Z. In practice, these may be files, records etc. A logical database state is an assignment of values to the logical data items

composing the database. Each logical dataitem may be stored at any DM in the system or redundantly replicated at several DMS. A stored copy of a logical dataitem is called a stored dataitem or simply a dataitem. A stored database state is an assignment of values to the stored dataitems in a database.

Transactions

Users interact with the DDBMS by executing transactions. A transaction is a sequence of operations on one or more data-items in order to change the state of the database. It is, in fact an on-line query expressed through application programmes written in a general purpose programming language.

An important property of the transaction is that it is atomic in nature. Thus, each transaction if executed alone on an initially consistent database, must terminate and must leave the database in a new consistent state.

System architecture

A DDBMS contains four components (Fig.3): transactions, TMS, DMS and data. Transactions communicate with TMS, TMS communicate with DMS, and DMS manage the data. TMS do not communicate with other DMS, nor do DMS communicate with other DMS.

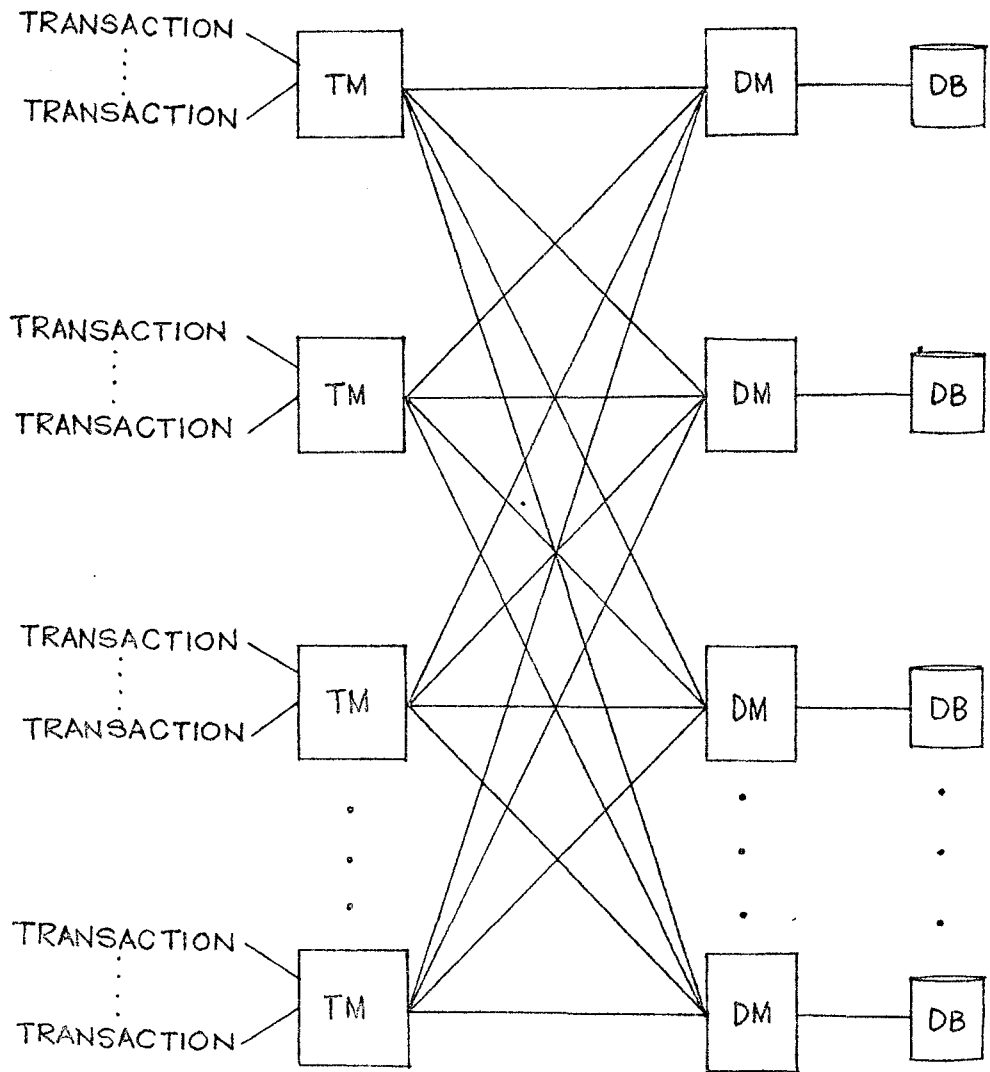


FIG.3 DDBMS SYSTEM ARCHITECTURE

TMS supervise transactions. Each transaction executed in the DDBMS is supervised by a single TM, meaning that the transaction issues all of its database operations to that TM. Any distributed computation that is needed to execute the transaction is managed by the TM.

Four operations are defined at the transaction-TM interface. Read operation retrieves the value of a dataitem from the database. Write operation writes into the database i.e. creates a new logical database state in which a dataitem has a new value. Begin and end operations are used to indicate starting and ending of transaction executions.

Several commercially available DDBMSs were developed by the vendors of centralized database management systems. They contain additional components which extend the capabilities of centralized DBMSs by supporting communication and cooperation between the DDBMSs which are installed at different sites of a computer network.

A DDBMS may be of two types depending on the local DBMSs used in the system : homogeneous and heterogeneous. In homogeneous DDBMS, each site has the same local DBMS, even if the computers and/or the

operating systems are not same. However, a heterogeneous DDBMS uses at least two different DBMSs in the system.

1.5 Distributed Database applications (SCHR 80)

The following systems make use of distributed database management systems.

Manufacturing control systems

These systems are structurally hierarchic. A central database is used for the overall scheduling and control of the manufacturing process and local databases, close to the process units, store only informations that is needed for supporting the local tasks.

Inventory systems

The inventory systems often present a hierarchic structure, with master stores and geographically distributed minor stores. The master stores may be connected through a generalized network and they can be the central nodes of star networks connecting the minor stores closed to each of them.

Some inventory informations (viz. quantities in local stores) are locally distributed without replication and heavy updating problems; however, other informations (viz. prices) are on the contrary replicated with full dependence.

Banking systems

In banking systems, there is a greater need to guarantee the database integrity than in the above mentioned systems. Therefore in banking systems the need is particularly felt for a central control, corresponding to a hierarchical architecture of the information system. Replicated information, such as personal accounts which are kept at the proper local agency and at the central agency, are periodically refreshed.

The developments in this field are expected to create the 'chequeless society', or even 'moneyless society', with computer communication between each purchase place and the buyer's personal account. For each purchase, the buyer's credit is checked, and, if permissible, his bank account is reduced and the seller's account increased.

Corporate database

A corporation represents an organization with many autonomous divisions, each of which can keep its own database. Some data of general interest can also be shared, typically, summary data can be maintained at high levels of the organization for strategical planning purposes.

Law enforcement systems

These systems include the information systems of the police, where data about criminals or terrorists are gathered. This kind of application seems naturally oriented towards distribution, since having the data available where they are needed is important; the data storage location is also distributed, since each police station usually keeps data belonging to its geographical area.

Medical systems

DDBMS may be applied to realize centralized hospital information systems, where data about patients' treatments can be stored.

Developments in this field would be able to provide a global architecture of medical information system which might consist of a general system connecting the computers storing the population's health databases, each of these computers would be the centre of a hierarchical system connecting the hospitals belonging to the same geographical area.

Public administration systems

These systems include demographic, fiscal, territorial information and other applications like managing of motor vehicle records or of telephone directories.

Increased population mobility makes the availability of demographic information necessary also in places other than the hometown, therefore the local demographic databases should be globally connected.

Fiscal information should also be supported by a distributed system in order to have a larger control upon everybody's activities.

Territorial information can be distributed in order to make urban or agricultural planning easier, by gathering local data and processing them.

CHAPTER - 2

CONCURRENCY CONTROL

This chapter has been devoted to the discussion on the problem of concurrency control in database systems. Section 2.1 introduces concurrency control with examples and section 2.2 presents a simple model of a DDBMS where steps of processing user interactions with the system have been discussed; in fact, this section helps in the understanding of the various steps that a transaction wishes to execute and presents a picture of an overall database management system where techniques to solve the problem of concurrency control are to be applied. Section 2.3 deals with the properties of valid transactions and the notion of database consistency that is to be maintained despite multiple access of the database by various transactions. Section 2.4 discusses the concept of serializability which provides a key to the resolution of the problem of concurrency control.

2.1 What is concurrency control

Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user database management system (DBMS). A multi-user database may be centralized or a distributed one,

In centralized database if the database is accessed by a single user, programmes accessing the database are run one at a time, thereby making the access serial; however, if it is accessed by more than one user, there is always a possibility that the database or more specifically a particular data item in the database may be accessed by some or all the user simultaneously and this simultaneous access is called concurrent access. Airline reservation system may be taken as an example (ULLM 84). It is a system with a centralized database where many sales agents may be selling tickets and changing lists of passengers and counts of available seats. If two or more agents run programmes to access the database, there is a possibility that a particular seat may be sold twice which is certainly an undesirable effect. Such problems arise due to concurrent access on database and is known as concurrency control problem.

In distributed database management systems (DDBMS), two or more users access databases stored at different sites of the network. For instance, in a banking system designed as a DDBMS, a particular account stored in some specified database may be required to be accessed by two or more users for retrieval or updating purposes. If a

TH-1762.



user's retrieval operation interfere with another's updating then the system would provide undesirable outputs to the users therefore, it is essential to prevent database operations performed by one user from interfering with operations performed by another and this is achieved by concurrency control. The problem of concurrency control in DDBMS is more complex than that in centralized database systems because (1) users may access data stored in databases of many different computers in a distributed system, and (2) a concurrency control mechanism at one computer can not instantaneously know about interactions at other computers (BERN 81).

Example of uncontrolled concurrent access

The following example illustrates two out of a number of ways in which users interfere because of uncontrolled concurrent access to databases.

Let there be an on-line electronic fund transfer system accessed by automated teller machines situated at remote sites to process the transactions. The transaction of a customer, requests for data retrieval followed by computations on the data and for storage of the result back into the database.

1) Suppose two customers simultaneously try to deposit money into the same account with a previous balance of

EXECUTION
OF T_1

DATABASE

EXECUTION
OF T_2

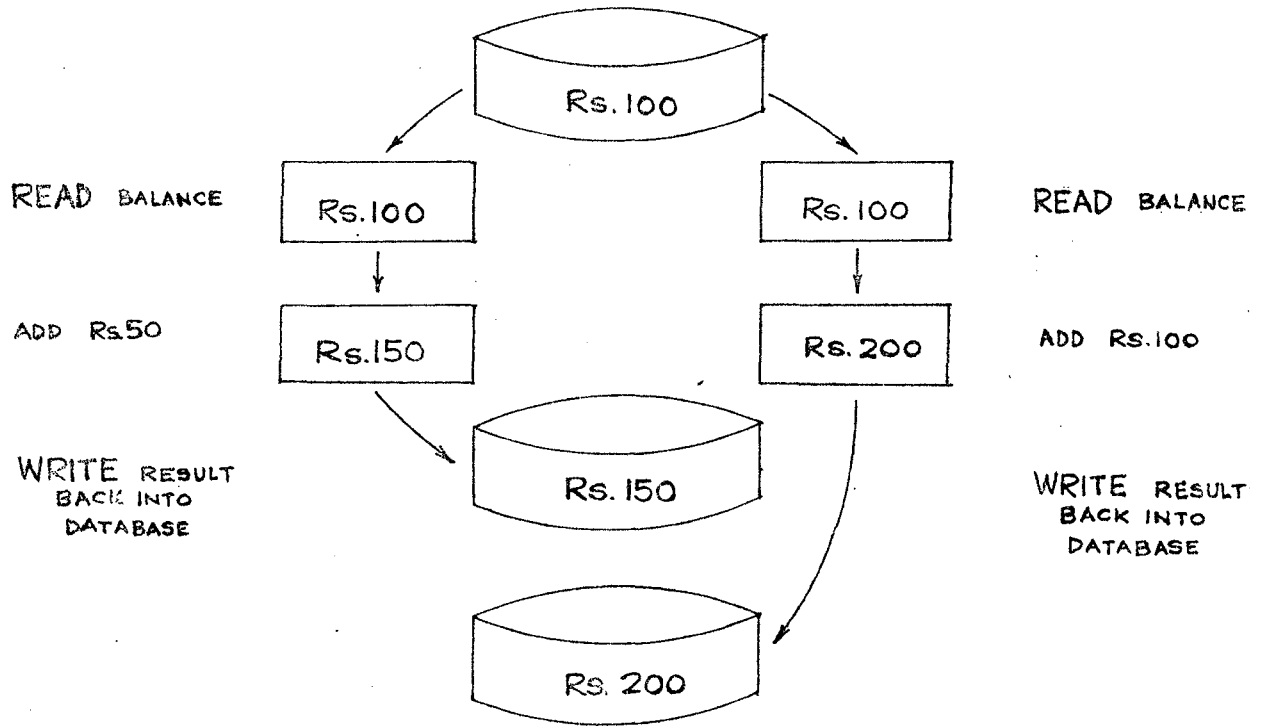


FIG.4 LOST UPDATE ANOMALY

Rs.100. The first customer deposits Rs.50 and the second deposits Rs.100 (Fig.4). The new balance in the account is the balance computed by either the first or the second customer depending on the order by which storing operation is executed. If first customer's storage operation precedes the second, the new balance in the account becomes Rs.200 ; otherwise, it remains as Rs.150. Thus the net effect of both the deposits on the database is incorrect; although two customers deposit money, the database only reflects one activity; the other deposit is lost by the system. This is a lost update anomaly because of concurrent execution of transactions.

2) Suppose two customers simultaneously execute the following transactions on a person's savings account and checking account. Originally, these accounts have Rs.200 and Rs.50 respectively (Fig.5).

Customer 1 : Transfer Rs.100 from the person's savings account to his checking account.

Customer 2 : Print the person's total balance in savings and checking account.

In the absence of concurrency control these two transactions interfere. The first transaction reads the savings balance, subtracts Rs.100 and stores the

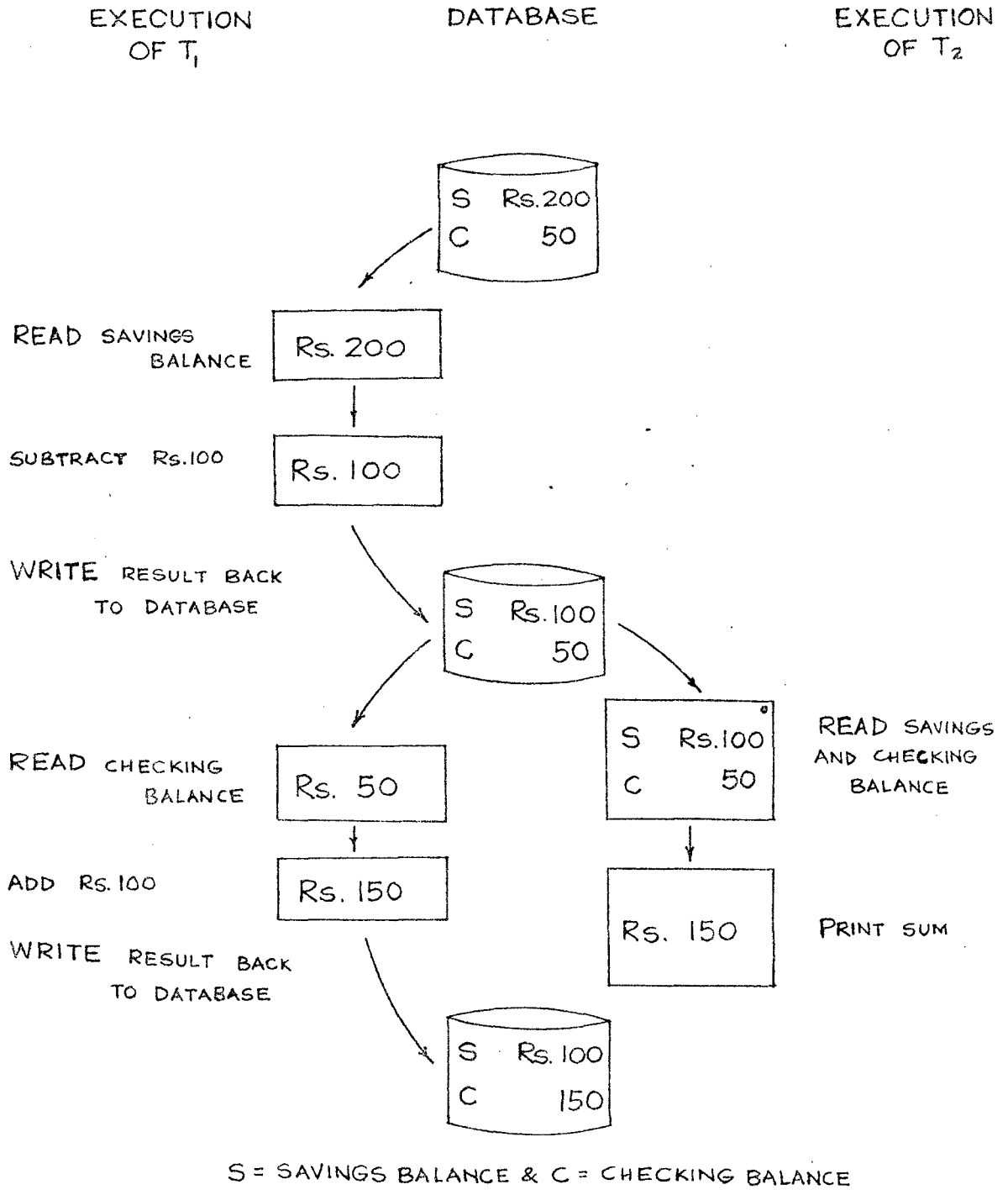


FIG.5 INCONSISTENT RETRIEVAL ANOMALY

result back in the database. Since the concurrency control is absent, the second transaction may start reading the savings and checking balance at this point and prints the total as Rs.150. Then the first transaction completes by reading the checking balance and then adding to Rs.100 and finally storing the result in the database. Unlike the previous case, the final values written into the database are correct; however, the retrieval by the second customer is incorrect which should have been Rs.250 instead of Rs.150. This is an inconsistent retrieval anomaly due to uncontrolled execution of concurrent transactions.

2.2 Transaction-processing model

Knowledge of the environment where transactions are processed is essential to understand the solution of concurrency control problem. The basic framework of transaction processing models, for centralized as well as distributed databases have been described in this section.

2.2.1 Centralized transaction processing model

A centralized DBMS has one transaction manager (TM) that supervises the transaction and one centralized database under the supervision of a data manager (DM). A transaction T accesses the DBMS by issuing the following

operations which are processed by TM.

BEGIN : By this operation, the TM sets a private workspace for the transaction where the workspace acts as a temporary buffer for values read from and written into the database.

READ (X) : When this command is issued, the copy of X is searched for by TM in the T'S private workspace. If the copy exists, it is used by T ; otherwise, the TM retrieves a copy of X from the database and gives it to T and puts it into T's private workspace.

WRITE (X, new value) : The TM again checks the private workspace for a copy of X and if it is found, the value is updated to new value. This 'write' operation does not store the new value into the permanent database.

END : The TM requests the DM to store back the updated value into the permanent database from the T'S private workspace. Then T finishes its execution and its private workspace is discarded.

Two-phase Commitment

Above steps are correct insofar as a transaction once started need not be aborted and restarted before the completion of its execution (aborting a transaction and restarting it by the system is essential in the

synchronization techniques discussed in later chapters). In case of restarting of a transaction, if the system requires the transaction to be aborted before all the involved data items in the database are updated to the new value, the database reflects the partial effect of the transaction and the effect is to be avoided.

Such partial effect can be avoided by requiring that each transaction either commits (the completion of a transaction is called 'commitment') by updating all the involved data items in the permanent database or does not, at all, start this updating. This property of transactions is called 'atomic commitment'. Two-phase commitment is a procedure to implement this property (LAMP 76, GRAY 78). Suppose a transaction T is updating data item X and Y. When T issues its END, the first phase of two-phase commitment begins, during which the DM issues the prewrite command that stores the values of X and Y from T's private workspace into secure storage. If the DBMS fails during this phase, no harm is done, since none of T'S update have yet been applied to the permanent database. During the second phase, the TM issues write command to DM to copy the values of X and Y.

into the stored database. If DBMS fails during the second phase, the database may contain incorrect information, but since the values of X and Y are already there on the secure storage, this inconsistency can be rectified when the system recovers.

2.2.2 Distributed transaction processing model

DDBMS has already been described in section 1.4 and it consists of more than one TM and DM and thus differs from centralized model in two aspects (BERN 81):

1) In centralized DBMS, it has been silently assumed that (i) private workspaces are part of the TM and (ii) data could freely move between a transaction and its workspace, and between a workspace and the DM. However, these assumptions do not hold good in case of DDBMS because TMS and DMS run at different sites and the movement of data between a TM and a DM may be expensive. These aspects relating to how T reads and writes data in the workspaces are studied under query optimization problem which has no direct effect on concurrency control.

2) The problem of implementation of two-phase commitment is complexified by the possibility that one site may fail while the rest of the system continues to operate. Because, if the failed site contains incorrect informations

in its database due to system failure, other sites may access those informations, thereby producing undesirable results. Thus the procedure for the implementation of atomic commitment of transactions is modified (the details of this procedure appear in HAMM 80).

In DDBMS, a transaction T accesses the system by issuing BEGIN, READ, WRITE and END operations. They are processed as follows :

BEGIN : A private workspace for T is created by the TM.

READ (X) : The TM checks T's private workspace to see if a copy of X is present. If so, that copy's value is made available to T. Otherwise the TM asks the DM to place the stored value of X in the workspace where it is received by T.

WRITE (X, new-value) : The value of X in T's private workspace is updated to new value, assuming the workspace contains a value of X.

END : When this operation is requested, two-phase commitment begins. For each X updated by T, and for each stored copy of x_i of X, prewrite (X_i) is issued to each DM where the copy is stored. This command copies the value of X from T'S private workspace onto

secure storage at respective sites. After all prewrites are processed, the new value is finally stored from the secure storage into the permanent database. Then T'S execution comes to end.

2.3 Transaction and Consistency

A transaction, a sequence of operations, is an atomic unit of database access, which is either executed or not executed at all and it has the following properties (CERI 84).

Atomicity : Either all the operations constituting the transaction are performed or none are performed. In case there is an interruption due to a failure, during the execution of operations, the partial results of already executed steps are rolled back and the original values of the affected dataitems prior to the beginning of the transaction are restored. Interruption of a transaction occurs because of two typical reasons :
(1) transactions abort for restarting purpose and
(2) system crashes.

Durability : Once a transaction commits, the system must guarantee that the results of its operations are never lost, independent of subsequent failures. The results preserved by the system are stored in the database.

Isolation : An incomplete transaction can not reveal its results to other transactions before its commitment. This property is needed in order to avoid the problem of cascading aborts (also called the domino effect) i.e. the necessity to abort all the transactions which have observed the partial results of a transaction that was later aborted. If, however, some of these transactions had already committed, we would have to undo already committed transactions, thus violating the transaction durability property.

Serializability : This is the most important property which provides the foundation for concurrency control and in fact concurrency control is the activity of guaranteeing transaction's serializability. If several transactions are executed concurrently, the result must be the same as if they are executed serially in some order.

Consistency (ESWA 76)

In database systems, users access shared data under the assumption that the data satisfies certain consistency assertions called consistency constraints. For example, let there be a banking system where there are two accounts with balances Rs.200 and Rs.300. If a transaction transfers money from one account to another,

the consistency constraint that the sum of the balances in both the accounts is Rs.500 is required to be satisfied.

If the values of the data items of a database satisfy the consistency constraints, the state of the database is called a consistent state. In fact, a valid transaction when executed alone, transforms the database into a new consistent state; that is, a transaction preserves consistency. Thus it can be relevantly concluded that a set of transactions if executed serially, also takes the database from a consistent state into a new consistent state.

2.4 Serializability

Serial execution of a set of transactions, is definitely a correct method for running concurrent transactions because it guarantees the database consistency. However, it prohibits the temporal interleaving of transaction steps and thus severely affects the performance by increasing the transaction-response time and reducing the system throughput (KOHL 81). Hence concurrent execution of transactions by interleaving the transaction steps is necessary for increasing the performance efficiently of the system; of course with

the condition that the execution of these steps preserves the consistency of the database. It may be noted here that transactions produce incorrect output if their steps are interleaved arbitrarily for concurrent execution which has been illustrated in the following example.

Example

Suppose in a banking system, there are three accounts A, B & C with balances Rs.200, Rs.100 and Rs.50 respectively. Two transactions T_1 and T_2 are required to be executed on them.

```
T1 : BEGIN
      READ ACC A obtaining A Balance
      READ ACC B obtaining B Balance
      WRITE ACC A as A Balance - Rs.100
      WRITE ACC B as B Balance + Rs.100
      END
```

```
T2 : BEGIN
      READ ACC B obtaining B Balance
      READ ACC C obtaining C Balance
      WRITE ACC B as B Balance - Rs.50
      WRITE ACC C as C Balance + Rs.50
      END
```

These two transactions T_1 and T_2 when executed, electronically transfer Rs.100 from ACC A to ACC B and Rs.50 from ACC B to ACC C respectively.

The consistency constraint in this case is that the sum of the account balances must be constant. If the transactions are run serially i.e. T_2 begins its execution after T_1 or T_1 begins its execution after T_2 , it is obvious that the consistency is maintained. However, if T_2 is allowed to run between the first and second write operations of T_1 , in the final state ACC A contains Rs.100, ACC.B contains Rs.200 and ACC.C contains Rs.100 with the sum of the balances being set to Rs.400 instead of Rs.350, which comprises an inconsistent state.

Thus it is necessary to provide a system with a mechanism that allows only those concurrent executions which are able to produce consistent database states. The correctness of the order in which the transaction steps are interleaved is determined by serializability of transactions.

2.4.1 Serializability in a centralized database

Let $R_i(x)$ and $W_i(x)$ denote read and write operations issued by a transaction T_i for the data item x . A sequence of operations performed by a set of

transactions form a schedule (also called as a history or log). For example, the following is a schedule for three transactions T_1 , T_j and T_k :

$$S_1 : R_1(x)W_1(y)R_k(x)R_j(x)W_k(y)W_j(y)$$

Two transactions T_1 and T_j execute serially in a schedule S if the last operation of T_1 precedes the first operation of T_j in S (or vice versa); otherwise, they execute concurrently. A schedule is said to be serial if no transactions execute concurrently in it. For example, the following schedule is serial :

$$S_2 : R_1(x)W_1(x)R_j(y)W_j(x)R_j(x)R_k(x)W_k(y)$$

In fact, a serial schedule defines an order among the transactions as in the case of S_2 , the order of operations indicates that $T_j(R_j, W_j, R_j)$ executes after $T_1(R_1, W_1)$ and $T_k(W_k, R_k)$ executes after $T_1(R_j, W_j, R_j)$. Hence the execution of a serial schedule is equivalent to the serial execution of the transactions forming the schedule.

However, if a schedule is concurrent (like S_1), their correctness is based on serializability :
 A schedule is correct if it is serializable, that is it is computationally equivalent to a serial schedule.

The term 'computationally equivalent' means if the execution of a schedule produces the same output and has the same effect on the database as that of some serial schedule, it is said to be computationally equivalent to the serial schedule. Since execution of serial schedules produces correct output and every serializable schedule is equivalent to a serial one, every serializable schedule is also correct.

After defining serializable schedule, it is required to develop a correct concurrency control mechanism which ensures that all executions are serializable or in other words the mechanism allows transactions to execute operations in such a sequence that only serializable schedules are produced.

In order to analyze the serializability of a schedule and correctness of concurrency control mechanism, we need the following two conditions which can be checked for determining whether two schedules are equivalent (PAPA 77, PAPA 79).

Condition 1 : Each read operation reads data item values that are produced by the same write operation in both schedules.

Condition 2 : The final write operation in each data item is the same in both schedules.

These conditions are applied in the analysis of concurrency control mechanism through the concept of conflicts between operations.

Two operations are said to be in conflict if they operate on the same data item, one of them is a write operation and they are from different transactions.

For example, $\langle R_i(x), W_j(x) \rangle$, $\langle W_i(x), W_j(x) \rangle$ are pairs of conflicting operations because each pair contains a write operation and also each operation in a pair operates on a single data item. $\langle R_i(x), R_j(x) \rangle$, $\langle W_i(x), W_j(y) \rangle$ are examples of nonconflicting operations since these requirements are not satisfied.

The condition for the equivalence of schedules can be restated by using the notion of conflicts in the following way :

Two schedules S_1 and S_2 are equivalent if for each pair of conflicting operations O_i and O_j such that O_i precedes O_j in S_1 , then also O_i precedes O_j in S_2 .

The following example shows how a schedule is checked

for serializability :

Example : Let there be two schedules S and S' represented by the following sequences of operations.

$$S : R_1(x) W_1(x) R_j(x) W_j(y)$$
$$S' : R_1(x) W(y) W_1(x) R_j(x)$$

These two schedules are equivalent because the unique pair of conflicting operations $\langle W_1(x), R_j(x) \rangle$ appears in the same order in both the schedules. The first schedule S is a serial schedule because the operations of the transaction T_1 precede all the operations of the transaction T_j . The second schedule S' is a serializable schedule for it is equivalent to serial schedule S.

The example also shows that in the serial schedule S, transaction T_1 precedes transaction T_j and this ordering of transactions is forced by the conflicting operations. Thus, in general it may be stated that precedence of transactions in the serialization order does not depend on the order of execution of the first operation of the transactions, but on the order of conflicting operations only (CERI 84).

2.4.2 Serializability in distributed database

In case of distributed database systems, there are a number of sites operating simultaneously. A

transaction introduced into the system at a site may require to perform operations at several other sites and in this way each site may have to process operations of several transactions concurrently. The sequence of operations performed by transactions at a particular site is called a local schedule. For example, if there are a distributed transactions T_1, T_2, \dots, T_n to be executed at m sites, then the execution is modeled by a set of local schedules S_1, S_2, \dots, S_m .

Ensuring serializability of a set of transactions in distributed systems is more complex because a local concurrency control mechanism applied at each node is not sufficient to guarantee the correctness of the execution of a set of distributed transactions. This has been illustrated in the following example.

Example

Let there be two transactions having following schedules under execution at two different sites :

$S_1(\text{Site 1}) : R_1(x)W_1(x)R_j(x)W_j(x)$

$S_2(\text{Site 2}) : R_j(y)W_j(y)R_1(y)W_1(y)$

These local schedules are individually serial; however there is no global serial sequence of execution of both transactions because in S_1 , transaction T_1 precedes

transaction T_j and in S_2 , transaction T_j precedes T_i . Thus a stronger condition than the serializability of local schedules is required to guarantee serializability of distributed transactions.

The execution of transactions T_1, \dots, T_n is correct if :

- 1) Each local schedule is serializable
- 2) There exists a total ordering of T_1, \dots, T_n such that, if T_i precedes T_j in the total ordering then there is a serial schedule S_k' such that S_k is equivalent to S_k' and all operations of T_i precede that of T_j in S_k' for each site K where both transactions have executed some action (CERI 84).

Papadimitriou et al. have expressed the above condition using the notion of conflicts in a proposition.

Proposition (PAPA 77, PAPA 79, STEA 76)

Let T_1, T_2, \dots, T_n be a set of transactions and let E be an execution of these transactions modeled by schedules S_1, \dots, S_m . E is correct (or serializable) if there exists a total ordering of such transactions for each pair of conflicting operations O_i and O_j from transactions T_i and T_j respectively, O_i precedes O_j in any schedule S_1, \dots, S_m if and only if T_i precedes T_j in the total ordering.

This proposition provides the foundation for devising a distributed concurrency control mechanism which would be correct if it allows only correct execution of distributed transactions. In other words, the mechanism has to guarantee that the conflicting operations for a set of transactions are processed in certain relative orders in order to attain serializability of execution of the transactions. An algorithm designed to maintain such order among the conflicting operations is called a synchronization technique to ensure correct execution of distributed transactions.

CHAPTER - 3

SYNCHRONIZATION TECHNIQUES BASED ON TWO-PHASE LOCKING & TIMESTAMP ORDERING

This chapter presents a description of the basic synchronization techniques developed for correctly executing concurrent transactions with maximal concurrency. Section 3.1 describes the technique of two-phase locking both for centralized and distributed database system and also discusses the management of deadlocks that arise in the implementation of the technique. In section 3.2, two types of timestamp ordering techniques have been presented : basic timestamp ordering and conservative timestamp ordering (CERI 84) Two other techniques like conflict graphs and reservation list have been described in chapter 4.

3.1 Two phase locking (2PL)

The synchronization techniques based on the approach of two phase locking have been discussed separately for centralized and distributed databases.

3.1.1 Case of centralized database

Whenever a transaction accesses a data item in a centralized database, it immediately locks it to prevent other transactions to access the same item during its own period of accession. In fact, in the simplest case

each data item has a unique lock which is held by at most one transaction at a time. However, if a transaction attempts to lock a data item that is already locked, it must either wait until the other transaction has released the lock or abort itself or pre-empt the other transaction. Each dataitem already locked and modified by an aborted or pre-empted transaction is restored to the state it was in prior to the transactions beginning and then it is unlocked. This operation preserves the consistent state of the database even if an incomplete transaction unlocks the dataitem.

There are two modes in which dataitems are locked :

- 1) A transaction locks a dataitem in shared mode if it wants only to read the dataitem .
- 2) A data item is locked in exclusive mode if a transaction wants to write into the data item.

Locking of a dataitem by shared and exclusive modes of more than one transaction is not arbitrary. Following rules govern the compatibility of lock-modes :

- 1) A transaction can lock a dataitem in shared mode if it is not locked at all or it is locked in shared mode by another transaction.
- 2) A transaction can lock a dataitem in exclusive mode only if it is not locked at all.

Conflicts

Two transactions are said to be in conflict if they want to lock the same data item with two incompatible modes :

1) If both the transactions attempt to lock on the same data item and one is applying readlock (i.e. shared mode) whereas other is applying writelock (i.e. exclusive mode), the resulting situation is known as shared-exclusive or read-write (rw) conflict.

2) If both the transactions attempt to lock on the same data item and one is applying writelock (i.e. exclusive mode) whereas other is also applying writelock (i.e. exclusive mode), the situation is called exclusive-exclusive or write-write (ww) conflict.

Synchronizations performed to avoid rw and ww conflicts are known as rw synchronization and ww synchronization respectively (BERN 81).

Correctness of 2PL mechanism (GERR 84)

Eswaran et al (ESWA 76) have proved that concurrent execution of transactions is correct if the following rules are observed :

1) Transactions are well-formed i.e. each of them always locks a data item in shared mode before reading it and always locks a data item in exclusive mode before writing it.

- 2) Compatibility rules for locking are observed.
- 3) Each transaction does not request new locks after it has released a lock. This means for each transaction, there is a first phase during which new locks are acquired (growing phase) and a second phase during which locks are only released (shrinking phase). In fact this condition names the mechanism as two phase locking.

During the shrinking phase, a transaction may release its exclusive locks at any time and this may allow other transactions to observe its result before its commitment; thus to avoid such an undesirable occurrence it is required that transactions hold all their exclusive locks until commitment.

Granularity of locking

In general, each transaction may lock data items of a database at record level or at file level. In the former case conflicts between transactions arise when two transactions want to access the same record. In the latter case, conflicts are instead determined when two transactions need to access the same file. Since the former case occurs with much less probability, locking at the record level allows more concurrency than locking at the file level. This aspect of relating the

size of the objects which are locked, with a lock operation is known as granularity of locking. It is preferable in DBMSs to provide locking at the record level (CERI 84).

In a centralized database, all transactions are performed according to the following scheme :

(Begin application)

Begin Transaction

Acquire locks before reading or writing

Commit

Release locks

(End application)

This scheme guarantees well-formedness and two-phasedness of transactions and consequently preserves the database consistency.

Deadlock

Deadlock is a major problem and can be illustrated by the following example (ULLM 84).

Example Suppose there are two transactions T_1 and T_2 whose locking and unlocking operations with two dataitems A & B are shown below (the main execution portions of the transactions have not been shown)

T ₁ :	LOCK	A	T ₂ :	LOCK	B
	LOCK	B		LOCK	A
	UNLOCK	A		UNLOCK	B
	UNLOCK	B		UNLOCK	A

Suppose T₁ and T₂ begin execution at about the same time. T₁ requests and is granted a lock on A and T₂ requests and is granted a lock on B. Then T₁ requests a lock on B, and is forced to wait because T₂ has a lock on that item. Similarly, T₂ requests a lock on A and must wait for T₁ to unlock A. Thus neither transaction can proceed; each is waiting for the other to unlock a needed item, so both T₁ and T₂ wait forever.

A situation in which each member of a set S of two or more transactions is waiting to lock an item currently locked by some other transaction in the set S is called a deadlock.

Following are the approaches made to resolve the deadlock problem in centralized databases :

- 1) Each transaction is required to request all its locks at once, and let the system grant them all, if the related data-items are not locked prior to the request made. Else, if one or more items are already locked by another transaction, the system does not grant the lock and the process is made to wait. In case of the above example,

the system grants locks on both A and B to T_1 , if it requests first and T_1 completes execution; then T_2 locks them and carries on the execution.

2) Another approach is to order the data items in an arbitrary manner and all transactions are required to lock them in this order. In case of the above example, if A precedes B in the ordering, then T_1 locks A before locking B; at this moment, T_2 would request a lock for A before B and would find A already locked by T_1 and would not be able to reach B. Thus B would be available to T_1 when requested by it. T_1 would complete and release the locks when T_2 could proceed. This approach can be shown to work perfectly in general case.

3) In this approach, transactions are allowed to run freely till the system discovers the deadlock. Deadlocks are discovered by waits for graphs. The graph contains nodes to represent transactions and arcs $T_1 \rightarrow T_2$ to signify that transaction T_1 is waiting to lock an item on which T_2 holds lock. If the system finds a cycle in such a graph, deadlock is detected; then it aborts and restarts one of the involved transactions and the effects of this incomplete transaction on the state of database is cancelled.

3.1.2 Case of distributed database

Implementation of 2PL mechanism in centralized database is easy because each dataitem exists as one copy only; consequently a transaction is able to discover a dataitem being locked by another transaction. However, data redundancy, necessary for reliability, availability and improved access time complexifies the implementation of 2PL mechanism in distributed database. This is because, two transactions which hold conflicting locks on two copies of the same dataitem stored at different sites could not know their mutual existence; and in such a case, locking of a dataitem becomes useless. Thus the implementation of 2PL in distributed database is performed in a different manner and four methods (BERN 81) for the purpose have been described below.

3.1.2.1 Basic 2PL Implementation

The basic 2PL is implemented by means of a 2PL scheduler which is a software module that receives the lock requests and lock releases and processes them according to 2PL specifications. These schedulers are kept distributed along with the database. For instance, the scheduler for dataitem X is placed at the site where X is stored. Two fundamental operations are required to

be performed on X :

1) To read X, a readlock (i.e. lock in shared mode) may be implicitly requested by read command on the data :

i) if the lock is granted by the scheduler, the read operation is carried on.

ii) otherwise, the request is placed on a waiting queue for the desired item till the item is free; after it is free the operation is carried on. Waiting may result in a deadlock and is resolved by methods described later in this chapter (Subsection 3.1.3).

By this reading operation, the required data X is retrieved from the database to the transaction's private workspace. The value of X is then updated to the new value at the workspace and then is to be written into the database from the workspace.

2) To write into X, writelock (i.e. lock in exclusive mode) may be implicitly requested by a prewrite command (not write command in order to achieve two phase commit) on the data :

i) if the lock is granted, the write operation is carried on.

ii) otherwise, it is made to wait in a queue till the item is free and then the required operation is

carried on. In case of a deadlock, it is resolved according to methods of subsection 3.1.3.

After an operation on a dataitem is over, corresponding locks are released by lock-release operations which are different for readlock and writelock. Then the operations on the waiting queue are processed in first-in/first-out order.

If basic 2PL is used for dealing with multiple copies of data, shared locks are acquired on one copy, while exclusive locks are acquired on all copies. That means for a logical dataitem X, having copies $x_1 \dots x_m$, a transaction may read one copy and need only obtain only one readlock on that copy; however while updating, it must obtain writelock on all copies of X.

3.1.2.2 Primary copy 2PL implementation

This technique pays attention to redundancy (STON 79). In this method of implementation one copy of each logical data item is named as the primary copy of that item. A transaction requiring the data item for its execution, obtains lock on it stored only in the primary copy of the item. All the read and write operations of the transaction are processed on that copy and then update messages are sent to other copies.

Read and write operations on a dataitem are processed in the following way; let x_1 be the primary copy of a dataitem X.

1) To read x_1 , some other copy of X, the site of the transaction communicates with the primary site as well as with the site that stores x_1 and readlock is acquired on x_1 at the primary site. If the lock is granted, item is read; otherwise, the request is made to wait till the item is free.

For readlock this technique requires more communication than basic 2PL; because in basic 2PL, data item is read from only the site where it is stored and consequently one message is sent, whereas in the primary copy 2PL, two messages are sent.

2) To write into X, a transaction issues prewrite commands to all sites where the data is stored but the writelock is requested on x_1 only. If the lock is granted, write command is executed and then update messages are sent to all copies, otherwise the transaction waits till the item is free.

For writelocks, primary copy 2PL does not require extra communication over the basic 2PL counterpart because write operations are similar except only that the writelock is obtained at a particular site.

3.1.2.3 Voting 2PL implementation

This approach exploits data redundancy and is due to Thomas (THOM 79). A transaction issues requests to all sites that hold a required data item. These sites acknowledge the receipt of the requests by saying "Lock set" or "Lock blocked" depending on whether the required item is locked or already under lock of some other transaction. The original site (i.e. where the said transaction originates) receives acknowledgements from other sites and count the number of lockset responses : if the number is strictly greater than the number of copies which are not locked, the site behaves as if all locks are set; otherwise, it waits for more lockset operations from sites that originally said "lock blocked" till the number of locksets become a majority. Because of waiting, there may arise deadlocks which can be resolved by techniques given in subsection 3.1.3.

- 1) To read X, a transaction requests readlocks on all copies of X. When a majority of locks are set, the transaction may read any copy.
- 2) To write into X, the concerned site sends prewrites to other sites with copies of X as a request for locks. When the majority of locks are granted to the transaction,

the site sends write request when the involved data item X is updated. Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be in its second commit phase at any time (BERN 81). All copies of X thereby have the same sequence of writes applied to them.

3.1.2.4 Centralized 2PL implementation

In this method of implementation one 2PL scheduler is placed at a single site unlike the previous methods where schedulers are distributed (ALSB 76a, GARC 79a). Here, appropriate locks are obtained from the central 2PL scheduler before accessing data at any site.

1) To read X from a site where X is not stored, the site first requests a readlock on X from the central site and waits for the central site to acknowledge that the lock has been set. Then the read request is sent to the site of X to read the data.

Since the lock is obtained in a round-about way, the communication is more than basic 2PL implementation and the cost of communication overhead thereby increases.

2) To write into X, a site issues prewrite request to the central site for a write-lock on X. After the lock

is obtained, it issues write request which is processed.

Here, the communication is also more for the same reason as above i.e. the prewrite does not request locks implicitly.

3.1.3 Management of distributed deadlocks

In distributed database management systems, deadlocks can arise in any of the preceding implementation of locking methods. The problem of deadlock resolution gets complexified in distributed systems because of the involvement of transactions originating from several sites.

Illustration

Suppose in a distributed database system, there are three sites S_1 , S_2 and S_3 with the following accounts

S_1 : ACC X	S_2 : ACC Y	S_3 : ACCZ
ACC Y	ACC Z	

Three transactions T_1 , T_2 and T_3 are executed respectively at S_1 , S_2 and S_3 .

T_1 : BEGIN ;	T_2 : BEGIN ;	T_3 : BEGIN ;
READ ACC X ;	READ ACC Y ;	READ ACC Y ;
WRITE ACC Y ;	WRITE ACC Z ;	WRITE ACCZ ;
END.	END.	END.

Let these transactions be executed concurrently with each transaction issuing its READ before any transaction issues its END. To preserve consistency the transaction would attempt to update all the copies of a particular dataitem. The transactions would proceed in the following steps :

Step 1 : T_1 obtains readlock on ACC X

T_2 obtains readlock on ACC Y

T_3 obtains readlock on ACC Z

Step 2 : T_1 requires writelocks on ACCY both at S_1 and S_2

T_2 requires writelocks on ACCZ both at S_2 and S_3

T_3 requires writelocks on ACCX at S_1 .

However writelocks would be obtained only after the readlocks are released i.e.

T_1 would not get writelock on ACCY at S_2 until T_2 releases the readlock on it and T_1 must wait.

T_2 would not get writelock on ACCZ at S_3 until T_3 releases readlock on it and T_2 must wait.

T_3 would not get writelock on ACCX at S_1 until T_1 releases readlock on it and T_3 must wait.

Thus, T_1 waits for T_2 , T_2 waits for T_3 which also waits for T_1 . In such a situation, transactions wait for locks which would never be available to them because a readlock would be released only after the completion of a transaction; but the completion is not possible and

deadlock results.

Deadlock situations can be characterized by waits-for graphs (HOLT 72, KING 74), which have been discussed in deadlock resolution in case of centralized databases (Section 3.1). The existence of a deadlock is concluded from the existence of a cycle in the waits-for graph. Figure 6 illustrates the deadlock situation of the above example.

Following techniques are available for resolving deadlock situation.

- 1) Time-out method
- 2) Deadlock prevention method
- 3) Deadlock detection method

1) Time-out method

With this method, a transaction is aborted after a given time interval has passed after the transaction enters a wait state. In fact, this method does not use waits-for graphs, but simply observes if any transaction waits for a dataitem beyond a specified time interval. If this interval passes away, the transaction is aborted and again restarted.

The main problem with timeout method is the choice of a good time interval. If the interval is longer,

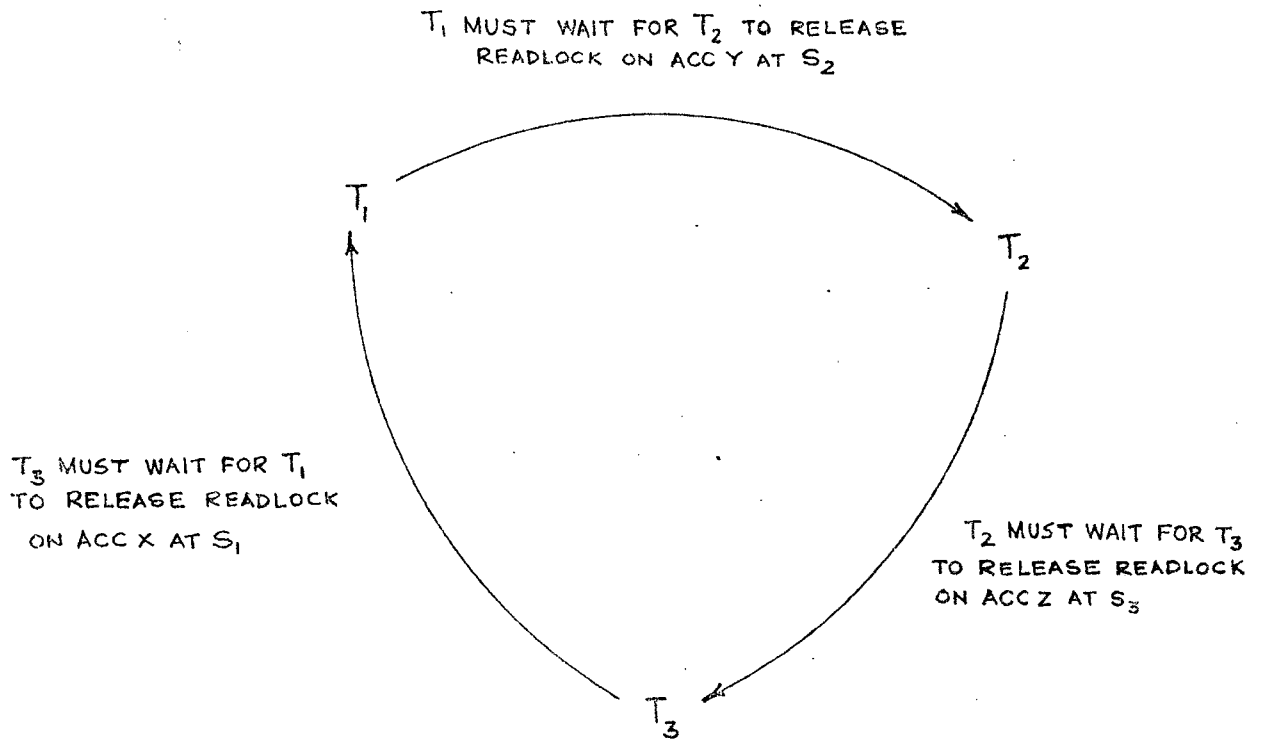


FIG.6 WAIT-FOR-GRAPH AND DEADLOCK SITUATION

then transactions would unnecessarily stay in deadlock before being aborted; if it is made shorter, transactions not in deadlock, but waiting for some data item would be unnecessarily aborted. That to, it is more difficult to choose a workable time interval in distributed systems than in centralized database because of the less predictable behaviour of the communication network and of remote sites.

Timeout method is acceptable for lightly loaded systems but not convenient for congested systems. Because in latter systems, short timeouts may induce cascading effect due to system overload. This happens when a transaction is aborted, not because it was in deadlock, but because the system was overloaded and therefore slow, leading to long waiting of the transaction. The abort operation causes additional delay due to additional message exchanges and work to be performed by the local systems. Such delays cause other transactions to be aborted and so on.

2) Deadlock prevention method

With the deadlock prevention scheme, a transaction is aborted and restarted if there is a possibility that deadlock might occur. Since related transaction is not allowed to wait for the concerned data item, the

possibility of occurrence of deadlock is totally eliminated.

Deadlock prevention is carried on in the following way : if a transaction T_1 issues a lock request for a dataitem which is held by another transaction T_2 , then a prevention test is applied; if the test indicates that there is a risk of deadlock, then T_1 is not allowed to enter a wait state. Instead, either T_1 is aborted and restarted, or T_2 is aborted and restarted. The previous algorithm is called nonpre-emptive and the second is called pre-emptive.

The prevention test must ensure that if T_1 is allowed to wait for T_2 then deadlock can never occur. This is obtained by arranging transactions in a particular order, like in order of their priorities. For two transactions T_i and T_j , T_i is allowed to wait for T_j only if T_i has got lower priority over T_j (If T_i and T_j have equal priorities, T_i can not wait for T_j , or vice versa. This test prevents deadlock because, for every edge $\langle T_i, T_j \rangle$ in the wait-for graph, T_i has low priority than T_j . Since a cycle is a path from a node to itself and since T_i can not have lower priority than itself, no cycle can exist.

In distributed systems, "timestamps" are used to decide the priority of transactions. Each transaction is assigned a unique number known as timestamp. The timestamp of a transaction consists of two parts : the local clock time at the beginning of the transaction read at the site of its generation and the unique site identifier which is appended to the clocktime at lower order bits (THOM 79). That the site does not send two transactions at the same local clock time is ensured by requiring that the site does not assign another timestamp until the next clock tick (BERN 81). Thus timestamps introduced into the system of different sites differ in their lower order bits (since different sites have different identifiers), while timestamps assigned by the same site differ in their higher order bits (since a particular site does not use the same clock twice). Hence timestamps are unique throughout the system and an old transaction has lower timestamps than young ones and intuitively they have higher priority as they are introduced to the system earlier than the young ones.

Two timestamp-based deadlock prevention schemes have been proposed in ROSE 78 :

Nonpre-emptive Method

If T_i requests a lock on a dataitem which is already locked by T_j , then T_i is permitted to wait only if T_i is older than T_j . If T_i is younger than T_j , then T_i is aborted and restarted with the same timestamp. Because, it is always better to restart the younger transaction. Therefore, in order to obtain a nonpre-emptive method, older transactions are allowed to wait for younger transactions which already hold a dataitem and younger transactions are not allowed to wait for older ones (CERI 84).

Pre-emptive Method

If T_i requests a lock on a dataitem which is already locked by T_j , then T_i is permitted to wait only if it is younger than T_j ; otherwise T_j is aborted and the lock is granted to T_i . In this method, the older transactions are allowed to pre-empt younger ones, and therefore only younger transactions wait for older ones.

The pre-emptive method may cause the following problem: suppose that T_j need to be pre-empted while it is in the second phase of two-phase commitment; in such a case T_j can not be aborted. This problem is resolved if T_i is not pre-empted; a deadlock does not

arise in such a case because a transaction which is in its second commitment phase can not be waiting for data items.

3) Deadlock detection method

With this method, transactions wait for each other in an uncontrolled manner and are only aborted if a deadlock actually occurs (BERN 81). In order to detect the deadlock, the system constructs global wait-for graph and searches for cycles. If a cycle is present, one of the transactions engaged in the deadlock is aborted, thereby breaking the deadlock. The aborted transaction is restarted and run to completion.

Construction of global wait-for graph is a major difficulty in distributed database systems though it is easy to construct local wait-for graph based on the wait-for relationships local to a particular site of the distributed system. Thus it is necessary to devise methods that efficiently combine the local wait-for graph into a global graph where the system would be able to search for a deadlock cycle. Two techniques have been explicitly described for the resolution of deadlocks by detection : centralized deadlock detection and hierarchical deadlock detection.

Centralized deadlock detection (GRAY 78, STON 79)

With the centralized method, each site is equipped with a local deadlock detector and a site is chosen at which a centralized or global deadlock detector is run. The local deadlock detector has the responsibility of discovering local deadlocks at the site concerned; however the centralized deadlock detector is responsible for building the distributed waits-for graph (DWFG) by collecting and connecting partial informations received from various sites and detects cycles in it. When a cycle is detected, the centralized detector selects the transactions to be aborted in order to break the deadlock situation.

Centralized deadlock detection is simple, but has two main drawbacks :

- 1) The detection operation may stop owing to the failures of the site where the centralized detector runs.
- 2) Building of DWFG at the centralized detector requires large communication costs in case of other sites of the network being located at far-off places. At times, it may so happen that a deadlock involves only a few sites which are close to one another, but for the construction of DWFG, those sites would have to

communicate with the distant centralized detector.

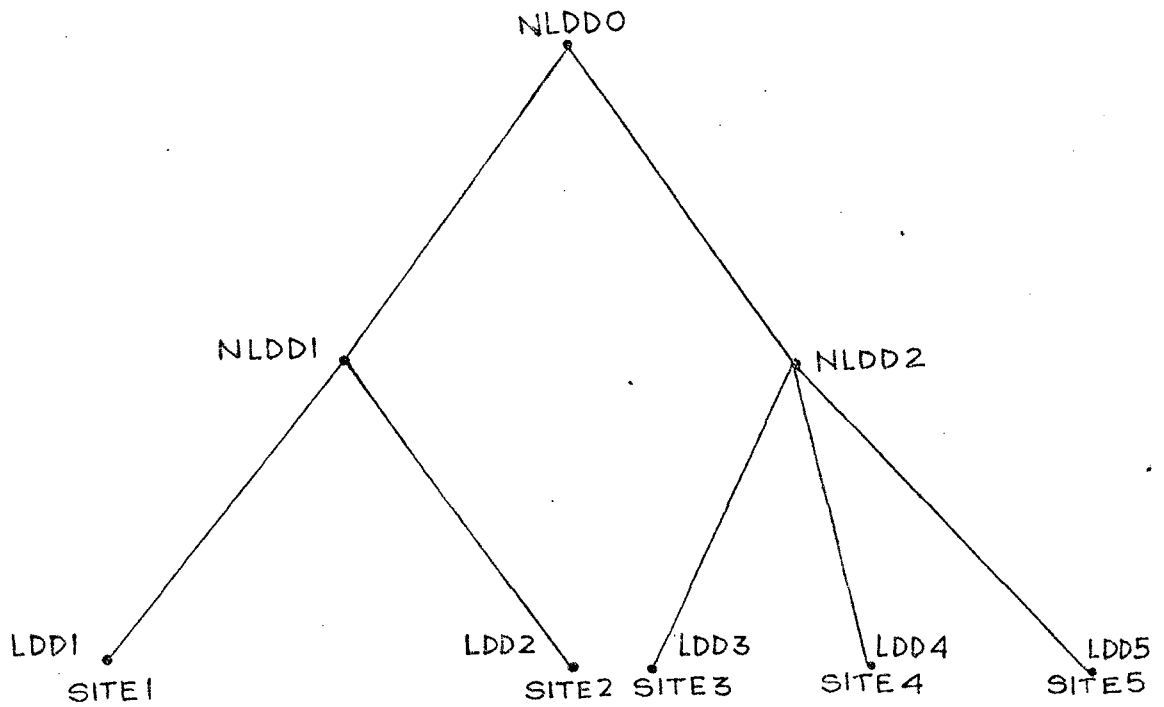
The hierarchical controller method resolves the problem of excessive communication cost.

Hierarchical deadlock detection (MENA 79)

With hierarchical method a tree of deadlock detectors is built, instead of having a set of local deadlock detectors and a single centralized detector. The detectors are arranged in a tree as shown in the figure (Fig.7). The local deadlock detectors (LDDs) are placed at the leaves of the tree whereas the nonlocal detectors are placed at non-leaf nodes.

Each local deadlock detector behaves like the local detector of the centralized method i.e. it determines local deadlocks and transmits information about global cycles to the nonlocal deadlock detectors at the immediately higher level in the hierarchy. Each of the nonlocal detectors detect deadlocks which involve only the deadlock detectors which are below it in the hierarchy.

In figure 7, LDD 1, LDD 2,LDD 5 are the local deadlock detectors situated at five sites. A deadlock involving site 1 and site 2 is detected at the immediately higher nonlocal detector i.e. NLDD 1; however, a deadlock involving site 1 and site 5 is detected only



NLDD = NONLOCAL DEADLOCK DETECTOR
LDD = LOCAL DEADLOCK DETECTOR

FIG.7 A TREE OF DEADLOCK DETECTORS

by NLDD) i.e. the highest level detector.

This approach of hierarchical detectors for detection of deadlocks is suitable for a group of sites where most of the database access request is within the group and few requests are sent to sites outside the group.

Disadvantages with detection method

Both the methods, centralized as well as hierarchical require that local waits-for informations be transmitted to one or more deadlock detector sites periodically. This periodic nature introduces two problems : firstly, a deadlock may prevail for several minutes without being detected, causing response-time degradation, secondly, a transaction T may be restarted for reasons other than concurrency control (like crash of the originating site) and in such a case some deadlock detector may find a cycle in the waits-for graph that includes T until T's restart propagates to the deadlock detector. Such a deadlock is known as phantom deadlock and when a detector finds a phantom deadlock it may unnecessarily restart a transaction other than T.

Another disadvantage with the method of deadlock detection is that restarting of partially executed

transactions increases the overall cost of the method. This cost is reduced by predeclaration where all the transaction's locks are obtained before its execution and consequently, the system only restarts those transactions that have not yet executed.

3.2 Timestamp ordering (T/O) Techniques

The timestamp ordering technique assigns a unique timestamp to each transaction to arrange the transactions in a sequential manner. A transaction that begins earlier has a smaller timestamp than a later transaction and hence precedes in that sequence. After timestamping, the transactions are processed so that their execution becomes equivalent to a serial execution in timestamp order. According to proposition of chapter 2, this means that conflicting operations get processed in the same order.

Conflicts are of two types depending on the kind of synchronization required. For rw synchronization, two operations conflict if (a) both operate on the same data item and (b) one is read operation and the other is write operation. For ww synchronization, two operations conflict if (a) both operate on the same data item and (b) both are write operations.

Below are described two timestamp ordering mechanisms: basic timestamp mechanism and conservative timestamp mechanism.

3.2.1 The Basic Timestamp Mechanism

The basic timestamp technique is implemented by building a scheduler, a software module that receives read or write operations according to timestamp specifications. The schedulers are distributed at various sites along with the database. The basic timestamp algorithm proceeds as follows :

1) A timestamp is assigned to each transaction when it is initiated at the site of origin. Each read or write operation which is required by a transaction has the timestamp of the transaction. Let this timestamp be TS .

For each data item X , let the largest timestamp (i.e. timestamp of the last transaction that has been processed on X) for read operation and write operation be $R-ts(X)$ and $W-ts(X)$ respectively. These timestamps are updated each time a transaction completes operation on this data item.

2) To avoid read-write conflict,

(a) the read operation of the current transaction with timestamp TS operating on data item X is :

- (i) rejected if $TS < W\text{-ts}(X)$ and the transaction is restarted with a new timestamp,
- (ii) executed otherwise; then $R\text{-ts}(X)$ is set to $\max(R\text{-ts}(X), TS)$.

(b) the write operation of the new transaction with timestamp TS on dataitem X is :

- (i) rejected if $TS < R\text{-ts}(X)$ and the transaction is restarted with a new timestamp,
- (ii) executed otherwise; then $W\text{-ts}(X)$ is set to $\max(W\text{-ts}(X), TS)$

3) To avoid write-write conflict, the write operation of the new transaction with timestamp TS on dataitem X is :

- (i) rejected if $TS < W\text{-ts}(X)$ and the transaction is restarted with a new timestamp,
- (ii) executed otherwise and $W\text{-ts}(X)$ is set to TS .

4) The restarted transaction, on assignment of a new timestamp which is certainly a larger timestamp is executed in accordance with rules (2) and (3).

The basic timestamp mechanism is deadlockfree, because transactions never wait: if a transaction does not execute an operation, it is restarted. That an operation can not be allowed does not depend on the fact that another transaction is momentarily operating on the same dataitem, but instead depends on the timestamp

associated with it. However, the deadlock freedom is a result obtained at the cost of restarting transactions.

Rules (2), (3) and (4) guarantee serializability because conflicting operations are executed in timestamp order at all sites and hence the timestamp order is the total order that makes the executions correct. However, above mechanism is integrated with two-phase commitment by using 'prewrite' operation to ensure that transactions are atomic.

Two phase commitment is incorporated by timestamping prewrites and accepting or rejecting prewrites instead of write operations. Once a scheduler accepts a pre-write, it must guarantee to accept the corresponding write no matter when the write request arrives. For rw (or ww) synchronization, once S accepts a prewrite (X) with timestamp TS it must not output any read (X) (or write (X)) with timestamp greater than TS until the write (X) is output. The incorporation is accomplished by substituting rules (2), (3) and (4) by the following :

2) Let TS be the timestamp of the prewrite operation P of a transaction on dataitem X. The operation is

- (i) rejected if $TS < R\text{-}ts(X)$ or $TS < W\text{-}ts(X)$

and the issuing transaction is restarted.

(ii) buffered along with its timestamp if
 $TS > R\text{-ts}(X)$ or $TS > W\text{-ts}(X)$.

3) Let TS be the timestamp of read operation R on data item X . The operation is

(i) rejected if $TS < W\text{-ts}(X)$

(ii) executed if $TS > W\text{-ts}(X)$

and only if there is no prewrite operation $P(X)$ pending on data item X having a timestamp $TS(P) < TS$.

(iii) buffered if there is one (or more) prewrite operation $P(X)$ with timestamp $TS(P) < TS$, until the transaction which has issued $P(X)$ commits. Buffering is necessary because, if executed, the write operation $W(X)$ corresponding to the prewrite $P(X)$ may be rejected by $TS(W) < R\text{-ts}(X)$

(iv) eliminated from the buffer after it is executed when no more prewrites with a smaller timestamp than R are pending on it.

4) Let TS be the timestamp of write operation on data item X . This is never rejected. But it has the possibility of being buffered if there is a prewrite operation $P(X)$ with a timestamp $TS(P) < TS$. The operation is otherwise executed and eliminated from the buffer.

The use of prewrites is equivalent to applying exclusive locks on dataitems for the time interval between prewrite and the commitment (write) or abort of the issuing transaction.

Thomas Write Rule

Let W be a write operation on dataitem X and suppose $TS(W) < W-ts(X)$. According to Thomas write Rule (TWR), the write operation W can be ignored instead of being rejected and restarted.

The rule works correctly because if $W_1(X)$ and $W_2(X)$ are two write operations such that $TS(W_1) < TS(W_2)$ then the execution of W_1 followed by W_2 is same as the execution of W_2 alone. Thus if W_1 is ignored and W_2 is executed, the final result obtained is same as if W_1 were executed before W_2 .

TWR applies to those write operations that try to place obsolete information into the database. For example, if we have a transaction that changes the price of a commodity, the new price is not a function of the previous price. If there is a correction on the previous price pending, we can simply ignore this correction after the new price has been written. The rule is also called "ignore-obsolete-rule".

The Conservative Timestamp Ordering Method

The conservative timestamping is a method for eliminating restarts by buffering younger operations until all older conflicting operations have been executed. Thus buffering is a part of the normal functioning of the method and helps in avoiding rejection of operations and restarting of transactions.

The conservative timestamping is based on the following requirements :

- i) Each transaction is executed at one site only and does not activate remote programmes.
- ii) A scheduler S_i must receive all the read requests (or write requests) from a different scheduler S_j in timestamp order. Since it is assumed that the network is a FIFO (First Input First Output) channel, this requirement is accomplished by requiring that schedulers send read requests (or write requests) to other schedulers in timestamp order.

Sending request messages in timestamp order can be implemented in two ways :

- i) It is possible to process transactions serially at each site. But this does not satisfy the purpose of concurrency control.

ii) Transactions can be executed by issuing all read requests before their main execution and all write requests after their main execution. For instance, if $TS(T_i) < TS(T_j)$, it is sufficient to wait to send R_j operations until all R_i operations have been sent and to wait to send the W_j operations until all W_i operations have been sent. Then the transactions execute concurrently.

The conservative timestamp algorithm proceeds as follows :

- 1) Each transaction is issued a unique timestamp when it is initiated at its site of origin. Each read or write operation which is required by a transaction has the timestamp of the transaction.
- 2) Read and (or) write request messages are sent to the site or sites containing the data item required by transaction in timestamp order.

Before going to next step, it is assumed that a site i has at least one buffered read and one buffered write operation from each other site of the network.

- 3) Read-write conflict is avoided in the following way :
 - a) For a read operation R that arrives at site i :
 - (i) if there is some write operation W buffered at site i such that

$$TS(R) > TS(W),$$

then R is buffered until these writes are executed,

(ii) otherwise, R is executed.

(iii) When R is buffered or executed, buffered operations are retested to see if they can now be executed.

b) For a write operation W that arrives at site i :

(i) if there is some read operation R buffered at site i such that

$$TS(W) > TS(R),$$

then W is buffered until these writes are executed.

(ii) otherwise, W is executed.

(iii) When W is buffered or executed, buffered operations are retested to see if they can now be executed.

4) To avoid write-write conflict for a write operation W that arrives at site i :

(i) if there is some write operation W' buffered at site i such that $TS(W) > TS(W')$ then W is buffered until these operations are executed,

(ii) otherwise, W is executed.

(iii) When W is buffered or executed, buffered writes are retested to see if they can now be executed.

Problems with Conservative Timestamp Ordering

Two phase commitment is not a problem in conservative timestamp ordering method because write operations are never rejected. However, above implementation suffers from the following problems :

1) If a site never sends an operation to some other site, then the assumption made in the above algorithm does not hold and the second site stops outputting. This problem is eliminated by requiring that each site periodically sends timestamped "null" operations to each other site. These operations have the sole purpose of conveying timestamp information and thereby unblocking real operations. Alternatively, blocked sites explicitly request for timestamped null operations from other sites.

2) Due to the buffering of read operation, the corresponding transaction is forced to wait and thus while implementing conservative timestamp technique, care must be taken to see that waiting does not result in a deadlock. The deadlock, if occurs, is avoided by sending null operations after a suitable timeout.

CHAPTER - 4

SYNCHRONIZATION TECHNIQUES BASED ON CONFLICT GRAPHS AND RESERVATION LISTS

More efficient synchronization techniques have been developed by refining the techniques of locking and timestamp ordering and by eliminating the problems inherent in them. Conflict analysis approach and reservation list approach are two such methods that work well in distributed database systems with different amounts of data redundancy (KOHL 81).

This chapter describes the methods of conflict analysis and reservation lists.

4.1 Conflict analysis

The method of conflict analysis is the synchronization technique used in SDD-1, a system for Distributed Databases, developed at the Computer Corporation of America (ROTH 80). The system consists of a collection of database sites interconnected through a communication network. Figure 8 shows configuration of the system consisting of three types of virtual machines : transaction modules (TMs), data modules (DMs) and a reliable network system (RelNet). Each site can contain either one or both types of

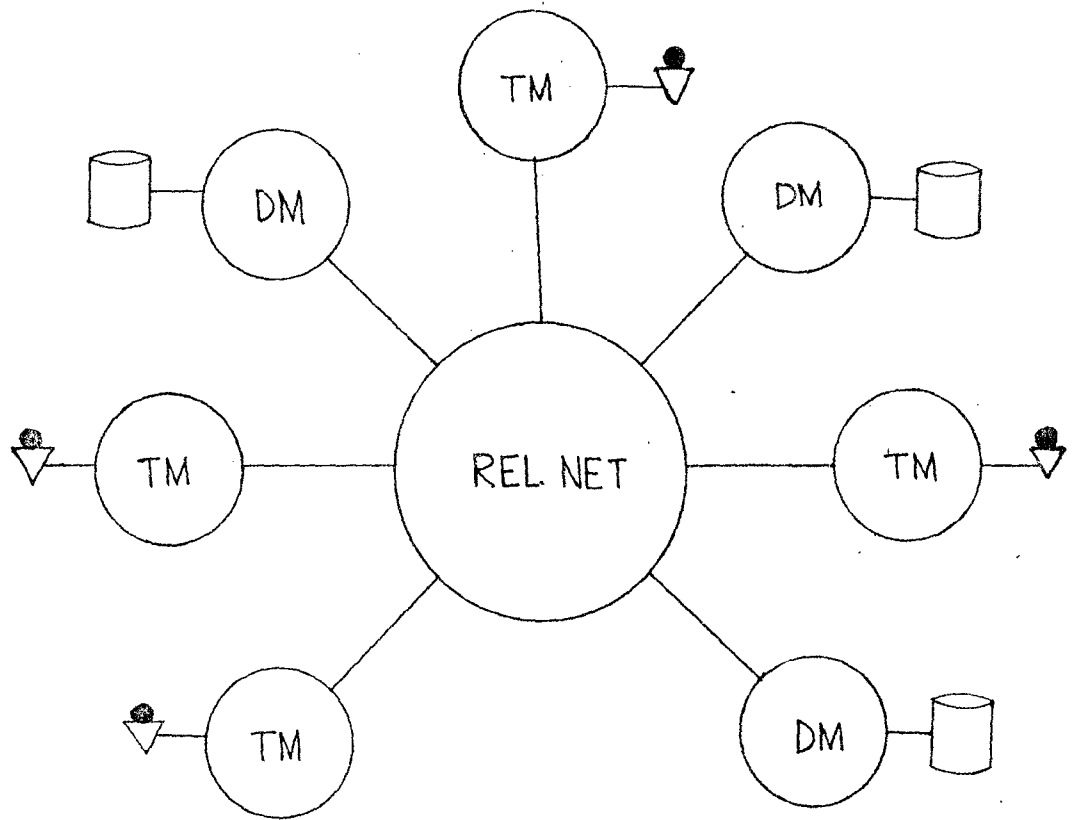


FIG. 8 SDD-1 CONFIGURATION

modules. DMS store physical data and behave much like conventional nondistributed DBMSs. TMs are responsible for supervising the execution of user transactions and they in fact, work as an interface between the user's perception of nondistributed data and the realities of data distribution and redundancy.

SDD-1 uses two mechanisms to ensure serializability of concurrently executed transactions. The first mechanism, called conflict graph analysis, is a technique for analyzing various "classes" of transactions to defect those transactions that require little or no synchronization (ROTH 80). Here, it is to be noted that SDD-1 mechanism does not assume that every transaction requires synchronization as strong as locking; because there exist transactions that some databases do not at all require synchronization even though they have overlapping write sets (BERN 80). The second mechanism consists of a set of synchronization protocols based on timestamps, which synchronize those transactions that need it.

4.1.1 Conflict Graphs

The database administrator defines "transaction classes" which are named groups of commonly executed

transactions at the time of system design. Each class is defined by its name, a read-set, a write-set and the TM at which it runs. A transaction is a member of a class if the transactions read-set and write-set are contained in the class's read-set and write-set respectively. The various transaction classes are not necessarily disjoint (R)TH 80). Conflict graph analysis (described below) is actually performed on these classes, not on individual transactions. Two transactions from different classes enter a conflict if their classes do so.

Example

Suppose there are three transaction classes defined by their read and write-sets :

C_1 : read-set = $\{a_2\}$, write-set = $\{a_2, c_3\}$

C_2 : read-set = $\{a_1, b_2, c_3\}$, write-set =
 $\{a_2, a_3, c_2, c_3\}$

C_3 : read-set = $\{a_1, b_2\}$, write-set = $\{a_2, c_2, c_3\}$

Let there be three transactions :

T_1 : read-set = $\{b_2\}$, write-set = $\{a_2, a_3\}$

T_2 : read-set = $\{a_1\}$, write-set = $\{c_2, c_3\}$

T_3 : read-set = $\{a_2\}$, write-set = $\{c_3\}$

Then it can be said that

T_1 is a member of C_2 ,

T_2 is a member of C_2 and C_3 ,

T_3 is a member of C_1 .

Conflict graph analysis is a technique to analyze the transactions on the basis of the predefined transaction classes in order to detect the presence of conflict. A conflict graph is an undirected graph that summarizes conflicts between transactions in different classes. For each class C_i , the graph contains two nodes, denoted r_i and w_i , which represent the read-set and write-set of C_i . The edges of the graph are defined as follows (Fig.9) : (1) For each class C_i , there is a vertical edge between r_i and w_i ; (2) for each pair of classes C_i and C_j (with $i \neq j$) there is a horizontal edge between w_i and w_j if and only if write-set (C_i) intersects write-set (C_j) ; (3) for each pair of classes C_i and C_j (with $i \neq j$), there is a diagonal edge between r_i and w_j if and only if readset (C_i) intersects writeset (C_j). The figure shows the conflict graph for the above example.

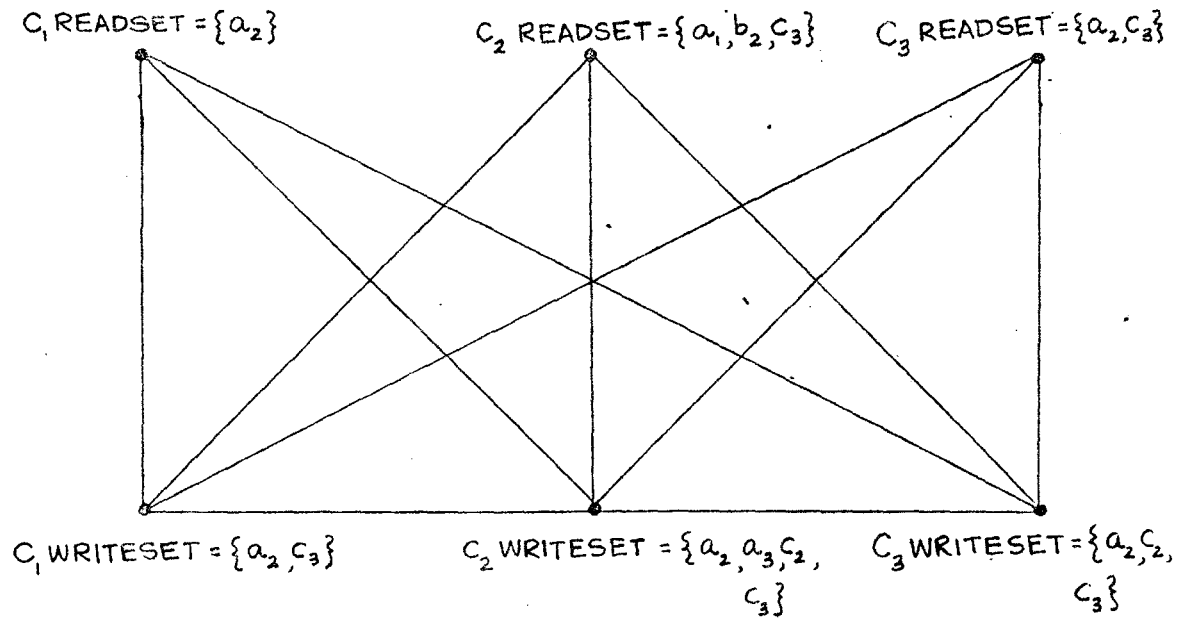


FIG.9 CONFLICT GRAPH

Different kinds of edges of a conflict graph (viz. horizontal edge or vertical edge) require different levels of synchronization. Synchronization as strong as locking is required only for edges that participate in cycles. Thus, in general, the output of analysis of conflict graph is a table that indicates

- i) for each class, which other classes it conflicts with and
- ii) for each such conflict, how much synchronization (if any) is required to ensure serializability (BERN 81).

It is assumed for convenience that each TM of SDD-1 is only permitted to supervise a particular class of transactions and vice versa. Thus, when a transaction T is submitted, the system determines the class to which the transaction belongs and sends it to the TM that supervises this class of transaction. The TM synchronizes the transaction against other transactions in its class using a local mechanism similar to locking. To synchronize the transactions against transactions in other classes, the TM uses the synchronization method (S) specified by the conflict graph analysis. These methods are called 'protocols'.

4.1.2 Timestamp-based protocols

SDD-1 uses four timestamp based basic protocols to synchronize transactions after the conflict graph indicates

the amount of synchronization required by each transaction. These are known as synchronization protocols and vary according to the degree of synchronization required and cost of use. For instance, the least expensive protocol is intended for transactions that can not interfere, such as reading the database to generate a sales slip through a point-of-sale terminal. The strongest and most expensive protocol is reserved for unanticipated transactions that are not known members of any of the predefined classes.

The rules that govern the selection of protocols for use in various situations determined on the basis of analysis of conflicts between transaction classes are known as protocol selector rules. The details of the protocols are complex and have been given along with the selection rules in BERN 80.

Bernstein et al. (BERN 79) have shown that the conflict analysis approach guarantees internal as well as mutual consistencies and allows more concurrency than the classical locking approach.

4.2 Reservation Lists

Milan Milenkovic' proposed a new reservation mechanism in 1979 for synchronizing concurrent updates

in distributed database systems that have high degree of data redundancy (MILE 79). The mechanism, a clever hybrid (KOHL 81) of locking and timestamp ordering is based on the use of a reservation list associated with every individually reservable database entity. The list contains an entry (viz. timestamp of the transaction) for each of the transactions that intends to use the related entity. Milenkovic' has devised algorithms to allow the transactions in the list to use one of two compatible synchronization protocols to update the entity. The use of reservation list ensures the maintenance of internal consistency of individual copies and mutual consistency of redundant copies in the database. This section presents an algorithm for a fully redundant distributed database system.

Assumptions

Following is the outline of underlying assumptions of the proposed solution.

Single-site access : It is assumed that all entities required by a transaction may be found at a single database site. This requirement is met when the entire database is replicated at several nodes and also met in some distributed database systems with partial redundancy.

Timestamps : Each transaction is assigned a timestamp by the database site where it enters the system. Timestamps are assumed to be unique and nondecreasing.

Message sequencing : All messages sent from one site to another are assumed to be delivered after a finite but variable delay in the same order as they were sent. No assumption about the relative ordering of the messages sent from two sites to a third one is made. This assumption is not fundamental to proposed solution but helps to eliminate numerous implementation details.

System's availability : All sites and communication channels are assumed to be available during the message exchange required by the algorithm. The synchronization protocols used by the transactions require regular message exchange between the database sites in the course of update processing.

Solution

The synchronization scheme for fully redundant distributed databases or a subset of the partially redundant database systems that satisfy the single site access assumption, allows transactions to use one of two compatible protocols. Protocol P (for pessimistic)

requires transactions to preclaim and reserve database objects prior to execution. But, under protocol O (for optimistic) the transaction is first tentatively executed, and the protocol subsequently checks whether the tentative update can be made permanent or must be rejected due to consistency conflicts. The protocols are designed to be compatible with each other so that they may be used in the same system concurrently and thus increase its flexibility.

Protocol P :

As described by Milenkovic', when a transaction T is submitted for execution to the database controller residing at a site S_i , the following set of rules constitutes protocol P :

1. Timestamping : S_i assigns a unique timestamp, derived from S_i 'S clock and unique identifier, to the transaction T, $TS(T)$.
2. Internal Reservation : S_i reserves entities from T'S read-set, $R S(T)$ and write-set, $WS(T)$, internally. If any of those entities is already reserved, T'S request is entered in the associated reservation list according to its timestamp : after all older reservation requests (whose timestamps are smaller than $TS(T)$), and in front of the younger ones.

3. Global Reservation : S_i broadcasts a reservation message on behalf of T . The reservation message has the following format :

Reserve ; sender's ID : S_i ,
timestamp : $TS(T)$,
identity of the entities to be
reserved : $RS(T)$

All reservation broadcasts sent by a site must be ordered according to the timestamp of the transactions themselves, i.e. they must contain increasing, although not necessarily consecutive timestamps.

4. Acknowledgements : Each site, upon receipt of the reservation broadcast, reserves entities contained in $RS(T)$ in its respective copy of the database according to rule 2. When this process is completed, an acknowledgement is returned to the site S_i .

Acknowledgement; responder's ID, transaction : T

5. Execution : Transaction T is executed, using S_i 's copy of the database, when the following conditions are met.

(a) An acknowledgement of the reservation broadcast is received from each database site, and

(b) T 's reservations become the oldest in S_i 's reservation lists associated with the entities contained in $RS(T)$ and $WS(T)$.

6. Update broadcast : S_i applies updates of T ($WS(T)$) to its copy of the database, removes internal reservations made on behalf of T , and broadcasts an update message to all other sites.

Update ; values of the entities to be
updated : $WS(T)$

7. Completion : Upon receipt of the update message, each site applies the specified updates to its copy of the entities contained in $WS(T)$, as soon as the reservations set on behalf of T become the oldest in the related reservation lists. Each such reservation is subsequently removed.

Because of the reservation scheme, the advantage with this protocol is that there is no need to reject concurrent updates involved in consistency conflicts. Each site is allowed to accept all the transaction load it can handle, because the existence of some transactions in progress does not prevent it from initiating the new ones, irrespective of whether they overlap or not.

Secondly, protocol P does not require permanent storage of timestamps because a timestamp ceases to exist when the associated transaction completes its execution.

Thirdly, the efficiency of the protocol is further enhanced because of relatively low delay and communication overhead due to the communication and reservation of only the writesets of updating transactions.

Lastly, timestamp ordering of transactions guarantees mutual consistency of all copies of the database and absence of deadlocks. The transactions generated in the system always complete execution in finite time because of the fact that a transaction may wait only for transactions that are older than it.

Protocol O :

Protocol O was designed to remove the restriction contained in protocol P that database entities must be preclaimed and reserved before the execution of a transaction can begin. This aims at improving efficiency for certain transactions that would otherwise have to reserve large portions of the database just to guarantee that all objects that are needed have been preclaimed.

Under the protocol O, after timestamping of a transaction T the entities required by it are locked locally at the initiating site and the transaction executed using these entities. All updates are

tentatively recorded by that site which then communicates with other sites in order to determine whether an older transaction executed elsewhere in the system obsoletes the work of T. Based on the information gathered from other sites, the initiating site decides whether to accept or reject the tentative updates of the executed transaction and announces its decision to the rest of the system.

According to Milenkovic (MILE 79) when a transaction is submitted to a database site S_i , the following steps constitute protocol O.

1. Timestamping : S_i assigns a unique timestamp to T - $TS(T)$.
2. Execution : S_i begins execution of T and locks each entity required by T. If an entity requested by T is not available, T is blocked and its request is entered in the associated lock/reservation list according to timestamp of T. If a younger transaction initiated by S_i owns the entity, the ownership is revoked and the said transaction is restarted. Restarting of a transaction consists of discarding of its tentative updates and releasing of its locks.

When all internal locks are granted to T, execution of T is completed and its updates are

tentatively recorded elsewhere in the database (but not in the database proper) by S_i .

3. Tentative update : S_i broadcasts a tentative update message on behalf of T . This message contains the values of the entities modified by T and its timestamp, $TS(T)$. All update broadcasts sent by a site must follow the timestamp ordering of the related transactions.

4. Acknowledgements and rejections : Each recipient of the tentative update broadcast records reservations for the entities contained in $WS(T)$ in its lock/reservation lists according to the timestamp of T . If a younger internal transaction owns some of the specified entities, it is restarted. If an update broadcast has already been sent on behalf of the restarted transaction, a reject message is sent to all other sites. This message will cause the reservations and the tentative updates of the restarted transaction to be discarded by all other sites as well. Following this process, each site acknowledges the receipt of broadcast to S_i .

5. Update broadcast : If T is still active when all acknowledgements are received, S_i makes the updates of T permanent. That is, S_i applies to its

copy of the database the values contained in WS(T) and broadcasts a make permanent message.

6. Completion : Each recipient of the make permanent broadcast makes the updates contained in WS(T) permanent in its copy of the database, as soon as the related reservations become the oldest in its lock/reservation lists. Such reservations are subsequently removed.

Under protocol O, transactions lock the required entities of the initiating site but reserve them at all other sites. The difference between lock and reservation is that locks do not have to be globally confirmed and, when granted, the related entities may be accessed and tentatively modified.

In this protocol, internal consistency is preserved by local locking of the entities when the transaction is under execution. But the mutual consistency among the copies of the database is maintained by rejection of obsolete updates as ensured by rule 4.

However, the protocol can not guarantee that transactions run under it are completed in finite time because of restarting of some transactions. Since a transaction is assigned newer timestamps each time it is restarted and

the number of restarts is unbounded, completion of such transactions in finite time is uncertain. This problem is resolved by keeping track of the number of restarts for each unsuccessful transaction. When this number exceeds a predefined limit, affected transaction is allowed to run under protocol P and thus completed in finite time.

Both the protocols described above are concerned only with updating transactions. Read only transactions can be executed in one of the following ways :

i) The system may regard the read only transaction as a "null update transaction" (empty write set) and run under protocol P or O, as appropriate. However, read only transaction executed in this way unnecessarily uses overhead of intersite communication and causes delay, though the transaction observes the consistent state of the database as of its timestamp.

ii) Entities required by a read only transaction are locked internally (in sharable mode) at the initiating site, and the local database copy is read when all locks are granted. In this way the transaction is guaranteed to see a consistent state of the database, but not necessarily as of its timestamp.

Comparision of protocols P and O :

- 1) Protocol P accepts and completes a transaction in finite time unlike protocol O.
- 2) Both the protocols require a comparable number of intersite messages and incur the same communication delay for accepted transactions.
- 3) Protocol O imposes higher storage requirements on the system, because all sites are supposed to keep the tentative updates until their fate is resolved.

Milenković states : "Protocol P should be used for transactions that are long and/or expensive to run, in order to avoid costly restarts. Protocol O on the other hand, should be used for transactions that are known to have low probability of conflicts, or for which preclaiming of resources is inefficient."

CHAPTER - 5

Integrated Concurrency Control

The previous two chapters describe the various techniques for synchronization of concurrently executed transactions in distributed database systems. The techniques of locking and timestamp ordering are the two basic approaches made to maximize concurrency while processing such transactions. Implementation of locking scheme is possible in different ways : basic two phase locking (basic 2PL), primary copy 2PL, voting 2PL, and centralized 2PL; that of timestamp ordering (T/O) is possible in ways like basic T/O, or conservative T/O. However, better implementation methods can be constructed by combining the above approaches.

A distributed database system experiences two types of conflicts : read-write and write-write. The techniques described in chapter 3 consist of using a particular type of implementation (either one of 2PL implementations or one of T/O implementations) for both the types of conflicts.. However, Berustein et al. (BERN 81) have suggested methods by using various types of implementations separately for rw and ww synchronizations.

5.1 Decomposition of Concept of Serializability

The serializability of execution of a set of transactions has been characterized in the proposition of chapter 2. In this proposition the two types of conflict (rw and ww) have been treated under the general notion of conflict; Bernstein et al. have decomposed this concept of serializability and restated the condition of serializability by distinguishing these two types of conflicts.

Let E be an execution of a set of transactions modeled by a set of schedules. The following binary relations, denoted by " \rightarrow " with various subscripts have been defined on transactions in E : for each pair of transactions T_i and T_j

- 1) $T_i \rightarrow_{rw} T_j$ if in some schedule of E, T_i reads some data-item into which T_j subsequently writes ;
- 2) $T_i \rightarrow_{wr} T_j$ if in some schedule of E, T_i writes into some data item that T_j subsequently reads ;
- 3) $T_i \rightarrow_{ww} T_j$ if in some schedule of E, T_i writes into some data item into which T_j subsequently writes ;
- 4) $T_i \rightarrow_{rwr} T_j$ if $T_i \rightarrow_{rw} T_j$ or $T_i \rightarrow_{wr} T_j$;
- 5) $T_i \rightarrow T_j$ if $T_i \rightarrow_{rwr} T_j$ or $T_i \rightarrow_{ww} T_j$

The relationship " \rightarrow " intuitively means "in any serialization must precede." For example, $T_i \rightarrow_{rw} T_j$

means " T_i in any serialization must precede T_j ".

Because, according to the proposition, if T_i reads x before T_j writes into x , then the hypothetical serialization in the proposition must have T_i preceding T_j .

Every conflict between operations in E is represented by an " \rightarrow " relationship. Therefore the proposition can be restated in terms of " \rightarrow ".

The proposition originally says that E is serializable if there is a total ordering of transactions such that for each pair of conflicting operations O_i and O_j from distinct transactions T_i and T_j (respectively), O_i precedes O_j in any schedule iff T_i precedes T_j in the total ordering. This latter condition holds if and only if the relation " \rightarrow " is acyclic. Such a relation is acyclic if there is no sequence $T_1 \rightarrow T_2, T_2 \rightarrow T_3, \dots, T_{n-1} \rightarrow T_n$ such that $T_1 = T_n$. Otherwise, the relation is cyclic and in that case it is meaningless to say that a particular transaction precedes another particular transaction. Hence the serializability of an execution of a set of transactions can be ascertained by knowing whether the relation " \rightarrow " is acyclic over these transactions. Bernstein et al. decomposed the relation " \rightarrow " into its components " \rightarrow_{rwr} " and " \rightarrow_{ww} " (according to the definition of binary relation

(5) above) and restated the proposition using them. The decomposed components are representatives of the read-write and write-write conflicts respectively.

Restated proposition (BERN 80a)

Let $" \rightarrow_{rwr}"$ and $" \rightarrow_{ww}"$ be associated with execution E. E is serializable if (a) $" \rightarrow_{rwr}"$ and $" \rightarrow_{ww}"$ are acyclic, and (b) there is a total ordering of transactions consistent with all $" \rightarrow_{rwr}"$ and all $" \rightarrow_{ww}"$ relationships.

This proposition is an immediate consequence of the first proposition (BERN 81) and indicates the following facts :

- 1) This way of characterizing serializability decomposes the problem of concurrency control into two parts : firstly, the relations $" \rightarrow_{rwr}"$ and $" \rightarrow_{ww}"$ must be acyclic and secondly, a total order among transactions is to be maintained in consistence with these relations in order to ensure serializability of the transactions.
- 2) The proposition implies that rw and ww conflicts can be synchronized independently except under the condition that there must be a total ordering of transactions consistent with both types of conflict. That rw conflicts are synchronized is ensured by the fact that $" \rightarrow_{rwr}"$ is acyclic and synchronization of ww

conflicts is ensured by the fact that " \rightarrow_{ww} " is keyclic. However, in addition to both the relations being acyclic, there must be a serial order consistent with all " \rightarrow " relations. In fact, this serial order integrates the two independent techniques and completes the solution of the problem of concurrency control in distributed database systems.

5.2 Integrated Concurrency Control Methods

Bernstein et al. show that the " \rightarrow_{rwx} " and " \rightarrow_{ww} " relationships are acyclic with respective techniques used and in addition, they provide an interface between the independent techniques. This interface, in fact, guarantees the total ordering of the involved transactions in confirmation with the condition of part (b) of Bernstein's proposition.

Various concurrency control methods have been listed that can be constructed using the different techniques of two-phase locking and timestamp ordering. For instance, a synthesis has been made between two-phase locking for rw synchronization and timestamp ordering for ww synchronization in order to construct a more efficient concurrency control method than a combination of a pure 2PL technique (or timestamp ordering technique) for both rw and ww conflict could

provide. The following is the description of a few of the proposed integrated methods.

Pure 2PL methods

These methods are results of the combination of different types of two phase locking techniques like basic 2PL, primary copy 2PL, voting 2PL and centralized 2PL. Three of these methods have been described below for illustration.

Each of these methods ensure serializability because each satisfies the required conditions : firstly, any two phase locking technique attains an acyclic " \rightarrow_{rw} " or " \rightarrow_{ww} " relation when used for rw or ww synchronization (BERN 79 b, ESWA 76, PAPA 79) which is a requisite condition according to the restated proposition; secondly, the total ordering of the transactions consistent with all " \rightarrow_{rw} " and all " \rightarrow_{ww} " relationships also exists and this order is the serialization order in which transactions obtain locks. This serialization order acts as the interface that binds together the independent techniques used for rw and ww synchronization. In addition to the interface, two-phasedness of the transactions need to be preserved i.e. while constructing integrated two-phase locking methods, it is to be seen that all locks needed for both

rw and ww techniques must be obtained before any lock is released by either technique (BERN 81).

Each of the above methods can be further refined by the choice of deadlock resolution technique as described in section 3.1.3.

Method 1 : Basic 2PL for rw synchronization and primary copy 2PL for ww synchronization.

In this method, a conflict between readlock and writelocks is resolved by basic 2PL technique, whereas, that between two writelocks by primary copy 2PL technique.

Suppose there is a logical dataitem X with copies x_1, \dots, x_m placed at various sites. If a transaction wants to read X it sends read command to any one site where a copy of X is stored. This command implicitly requests a readlock on the copy of X at that site. To write into X, the transaction sends prewrite commands to every copy of X and the commands implicitly request writelocks on the copies. Bernstein et al. classify the writelocks into three types due to the fact that various types of writelocks need to be obtained at various copies for the locking conflict rules vary for writelocks from copy to copy of a dataitem.

- (i) Rw writelock : such a writelock only conflicts with readlock.
- (ii) Ww writelock : such a writelock conflicts with another similar writelock.
- (iii) Rww writelock : Such a writelock conflicts with readlocks, ww writelocks and also rww writelocks.

While using basic 2PL for rw synchronization, a transaction willing to read a data item X requests for readlock on any copy of X. This readlock conflicts with writelocks on all copies when another transaction is willing to write into X and that the prewrite of this transaction attempts to obtain rw writelock (as this writelock only conflicts with a readlock on the same data item) on all the copies of X. Thus this type of rw conflicts may be resolved at all copies.

On the contrary, writelocks conflict with another writelock only in the primary copy and thus it is resolved only at that copy. Since a readlock can also be obtained at the primary copy, the write-lock to be used here should be rww type.

Method 2 : Primary copy 2PL for rw synchronization and voting 2PL for ww synchronization.

In this method, read-write conflicts are resolved in the primary copy only whereas the write-write conflicts

are resolved by requiring that a transaction can write into a particular data item only when the system grants majority of writelocks to the transaction.

Suppose there are copies x_1, \dots, x_m of a logical data item X and x_1 is the primary copy of X . To read X , a transaction sends read request which implicitly obtains readlock on the primary copy x_1 . Once the readlock is granted, the transaction can read any copy of X . However, a transaction willing to write into X , first sends prewrite commands to each site that stores a copy of X . The prewrite command at the primary copy obtains a rw writelock which prevents other readlock requests from accessing the item. Thus read-write conflicts are resolved at the primary copy.

The ww synchronization is obtained by voting 2PL technique. When a transaction issues prewrites in order to write into x_i , all prewrite (x_i) commands except prewrite (x_1) request for a rww writelock on the primary copy of X i.e. x_1 where read commands in general are allowed to obtain readlocks. If the rww writelock can not be set on this copy, an rw writelock is set on x_1 before rww writelock is made to wait. A transaction writes into every copy of the required dataitem if it is granted a ww (or rww) writelock on majority of copies of it.

Method 3 : Centralized 2PL for rw synchronization and basic 2PL for ww synchronization.

Suppose there are copies of a logical data item X residing at various sites. Since centralized 2PL technique is used for rw synchronization, a transaction before reading (or writing) any copy x_i of X, obtains a readlock (or rw writelock) on X from a centralized 2PL scheduler.

Since the basic 2PL is used for ww synchronization, before writing X, a transaction sends prewrites to every site that stores a copy of X and these prewrites implicitly request ww writelocks on every copy of X. When two such ww writelocks enter a conflict, one is processed and the other one is made to wait which is processed after the first one releases the lock.

In all the above methods, readlocks are explicitly released by special lock release commands while writelocks are implicitly released by write commands (because prewrite command sets a writelock, after required computations is performed, the data in the original database position is updated in a two phase commitment manner where the write command takes the updated data into the database by simultaneously releasing the writelock on it).

Pure Timestamp Ordering (T/O Methods)

The basic T/O technique, the conservative T/O technique and Thomas write Rule (for ww synchronization only) can be combined to form various integrated methods.

These methods guarantee serializability as each of them satisfy the required conditions : firstly, the technique of timestamp ordering attains an acyclic " \rightarrow_{rwr} " or " \rightarrow_{ww} " relation when used for rw or ww synchronization ; this is because each site processes conflicting operations in timestamp order and thus each edge of the " \rightarrow_{rwr} " or " \rightarrow_{ww} " relation is in timestamp order ; since all transactions have unique timestamps, no cycles are possible. Secondly, the total ordering of the transactions consistent with all " \rightarrow_{rwr} " or " \rightarrow_{ww} " relationships also exists and the timestamp order is the valid serialization order that satisfies the restated proposition.

Because two different T/O techniques are used independently for rw and ww synchronization, the interface between the techniques is maintained by requiring that both techniques use the same timestamp for any given transaction.

Three of the integrated methods have been described for the sake of illustration.

Method 1 : Basic T/O for rw synchronization and conservative T/O for ww synchronization.

A transaction is assigned a globally unique timestamp which is used for both rw and ww synchronization. Each data item is associated with a read timestamp $R - ts$ and a write timestamp $w - ts$. In order to achieve two-phase commitment of a transaction, its read and prewrite commands are buffered.

Let $\min - R - ts(x)$ and $\min - P - ts(x)$ be the minimum timestamps of any buffered $read(x)$ and $prewrite(x)$ commands on data item x . Suppose R denotes the $read(x)$ command and P denotes a $write(x)$ command.

The steps for the method would be as follows :

- 1) If $ts(R) < w - ts(x)$, R is rejected.
else, if $ts(R) > \min - P - ts(x)$, R is buffered.
else R is output and $R - ts(x)$ is set to $\max(R - ts(x), ts(R))$.
- 2) Since conservative T/O is used for ww synchronization, a prewrite command is always buffered instead of being rejected.
- 3) If $ts(w) > \min - R - ts(x)$ or if $ts(w)$ is greater than the minimum timestamp of any buffered write command from some transaction site, W is buffered. Else W is output and $w - ts(x)$ is set to $ts(w)$.
- 4) When W is output, its prewrite is debuffered and the buffered read and write commands are retested to

see if any of them can be processed for an output.

Method 2 : Basic T/O for rw synchronization and
TWR for ww synchronization.

Here also, each dataitem is associated with read and write timestamps as in the previous case, however, the steps of the method are different and are as follows :

- 1) If $ts(R) < W-ts(x)$, R is rejected. Else if $ts(R) > min-R-ts(x)$, R is buffered. Else R is output and $R-ts(x)$ is set to $\max (R-ts(x), ts(R))$.
- 2) If $ts(w) > w-ts(x)$, the write command is processed as usual i.e. x is updated. If, however, $ts(w) < w-ts(x)$, W is ignored according to TWR and it has no effect on the database. In this method, a scheduler always accepts prewrite commands but never buffers write commands.
- 3) When W is output, its prewrite is debuffered and the buffered read commands and the write commands (if any) are retested to see if any of them can be processed for an output.

Method 3 : Conservative T/O for rw synchronization and TWR for ww synchronization.

In this method, each data item is required to be associated with a read timestamp and a write timestamp which are the timestamps of the respective operations that have already been processed on the data item. Let $\text{Min-W-ts}(S_i)$ be the minimum timestamp of any buffered write command from a site S_i . Let the read command of a transaction to be executed on a data item X be denoted by R , a prewrite command by P and a write command by W . Then the method consists of following steps :

- 1) If $\text{ts}(R) > \text{min-W-ts}(S_i)$ for any S_i , R is buffered; else it is output.
- 2) A prewrite command is always buffered till the write command arrives and if $\text{ts}(w) < w\text{-ts}(X)$, W has no effect on the database; that is such write command is ignored. Else, if $\text{ts}(W) > w\text{-ts}(X)$, it is output.
- 3) When W is output, its prewrite is debuffered; buffered read commands and of course, the incoming write commands (if corresponding prewrite command is buffered) are retested to see if any of them can be allowed to operate on the data.

Mixed 2PL and timestamp ordering methods

These methods are constructed by using two-phase locking technique for rw (or ww) synchronization and timestamp ordering technique for ww (or rw) synchronization. However, in order to guarantee serializability

of executions, the methods must satisfy both the conditions stated in the theorem :

1) The relation " \rightarrow_{rwr} " and " \rightarrow_{ww} " are required to be acyclic, which is of course the case with each of 2PL and timestamp techniques.

2) It is required that there is a total ordering of transactions consistent with all " \rightarrow_{rwr} " and all " \rightarrow_{ww} " relationships. This condition requires an interface to be built between the independent techniques and that the interface is required to guarantee that the combined " \rightarrow " relation (i.e. $\rightarrow_{rwr} \cup \rightarrow_{ww}$) remains acyclic. This means, the interface must ensure that the serialization order induced by rw technique is consistent with that induced by the ww technique. The interface given below makes this guarantee.

The interface

In any 2PL technique, a transaction owns all the locks it will ever own at the end of its growing phase (the first phase of two-phase commitment discussed in chapter 2), known as the locked point of the transaction. Then, in a serial execution it is a fact that all transactions start their execution at their respective locked points and also that is the case with all serializable executions. Hence these locked points of an

execution determine the serialization order of the execution. However, the serialization order induced by any timestamp ordering technique is obviously determined by the timestamps of synchronized transactions. There being different serialization orders for different techniques, if one technique is used for rw synchronization and another for ww synchronization, there would be problem of total ordering among the transactions. This problem is resolved by requiring the locked points to induce timestamps of the transactions (BERN 80b).

Locked points induce timestamps in the following way. Each data item X of a database is required to be associated with a lock timestamp, $L\text{-ts}(X)$. When a transaction T sets a lock, it simultaneously retrieves $L\text{-ts}(X)$. When T reaches its locked point, it is assigned a timestamp, $ts(T)$, greater than any $L\text{-ts}$ it retrieved. When T releases its lock on X , it updates $L\text{-ts}(X)$ to be $\max(L\text{-ts}(x), ts(T))$.

It can be proved that timestamps generated in this way are consistent with the serialization order induced by 2PL technique i.e. $ts(T_j) < ts(T_k)$ if T_j must precede T_k in any serialization induced by 2PL.

Proof

Let T_1 and T_n be a pair of transactions such that T_1 must precede T_n in any serialization.

Thus there exist transactions $T_1, T_2, \dots, T_{n-1}, T_n$ such that for $i = 1, \dots, n-1$

- (a) T_i 's locked point precedes T_{i+1} 's locked point and
- (b) T_i releases a lock on some data item X before T_{i+1} obtains a lock on X .

If L is the L -ts(X) retrieved by T_{i+1} , then $ts(T_i) < L < ts(T_{i+1})$ and by induction $ts(T_1) < ts(T_n)$.

Therefore, timestamps generated are consistent with the serialization order induced by 2PL (BERN 81).

A mixed method using basic 2PL for rw synchronization and Thomas Write Rule (TWR) for ww synchronization has been described below for the sake of illustration.

Method

This method requires that every stored data item have a lock timestamp L -ts and a write timestamp W -ts.

Let X be a logical data item with copies x_1, \dots, x_n . To read X , a transaction T issues read command on any copy of X , say x_i . This command implicitly requests a readlock on x_i and when the readlock is granted, L -ts(x_i) is returned to T .

To write into X , T issues prewrite commands on every copy of X . These command request writelocks (or more specifically known as rw writelocks that only conflict with readlocks on X) on the corresponding copies, and as each

writelock is granted, the corresponding L-ts is returned to T. When T reaches the locked point i.e. when all the required locks are obtained $ts(T)$ is calculated as described in the last section. This timestamp is assigned to the write command which are then sent for updating purpose.

These write commands are processed using Thomas Write Rule. Let W be the write command to update x_j :

- i) if $ts(W) > W-ts(x_j)$, the write command is processed as usual and consequently x_j is updated.
- ii) if $ts(W) < W-ts(x_j)$, W is ignored.

This method has the advantage over pure 2PL methods in the sense that here transactions execute concurrently even if their write-sets intersect. This is because, writelocks never conflict with other write-locks and those obtained by prewrites are used only for rw synchronization. Also, the write command of a transaction is processed only after it is assigned a timestamp which is induced by the locked point of the transaction i.e. after the transaction obtains all of its locks.

5.3 Conclusion

This dissertation on synchronization techniques does not contain all the techniques available in literature, but concentrates on the basic frameworks

required to achieve maximal concurrency while running concurrent transactions.

All of the concurrency control mechanisms discussed in last three chapters have been designed for use in a distributed transaction-processing environment. The data-items have been assumed to be independent database entities directly associated with physical or logical storage units (pages, records, or files). Transactions have been assumed to consist of a sequence of read and write operations and of course, with local computations. Thus, the techniques are not able to support concurrency control in a general distributed environment; however, they provide a general framework that resolve the problems arising out of multiple access to a shared data where database consistency need to be preserved.

Although various techniques have been developed for synchronization of concurrent execution of transactions, performance of a few of a them has been evaluated. Factors influencing the performance when the techniques are used, are system throughput and transaction response time which are under the influence of intersite communication, local processing, transaction restarts and transaction blocking. The impact of these

factors varies from technique to technique (BERN 81).
Thus, a comprehensive analysis and comparison between
the various techniques need to be studied in order to
optimize their use in distributed database systems.

BIBLIOGRAPHY

- ALSB 76 ALSBERG, P.A., and DAY, J.D. "A principle for resilient sharing of distributed resources", in Proc. 2nd Int. Conf. Software Eng., Oct. 1976, pp. 562-570.
- BERN 79 BERNSTEIN, P.A., SHIPMAN, D.W., and WONG, W.S., "Formal aspects of serializability in database concurrency control", IEEE Trans. Softw. Eng. SE-5, 3(May 1979), pp. 203-216.
- BERN 80 BERNSTEIN, P.A., SHIPMAN, D.W., and ROTHNIE, JR. "Concurrency control in a system for distributed databases (SDD-D)", ACM Trans. on Database Systems 5, 1(March 1980), pp. 18-51.
- BERN 80a BERNSTEIN, P.A., and GOODMAN, N. "Timestamp based algorithms for concurrency control in distributed database systems", Proc. 6th Int. Conf. Very Large Data Bases, Oct. 1980.
- BERN 80b BERNSTEIN, P.A., GOODMAN, N., and LAI, M.Y. "Two Part Proof Scheme for Database Concurrency Control", in Proc. 5th Berkeley Workshop Distributed Data Management and Computer Networks, Feb, 1980.
- BERN 81 BERNSTEIN, P.A., GOODMAN, N. "Concurrency Control in Distributed Database Systems", in Computing Surveys 13, 2 (June 1981), pp. 185-221.
- CERI 84 CERI, S., PELAGATTI, G. "Distributed Databases Principles and Systems" - Mc Graw-Hill Book Company, 1984.
- DAVI 81 DAVIES, D.W., HOLLER, E., JENSEN, E.D., KIMBLETON, S.R., LAMPSON, B.W., LELANN, G., THURBER, K.J., and WATSON, R.W. "Distributed Systems - Architecture and Implementation Vol.105, Lecture notes in Computer Science, Berlin, Springer-Verlag, 1981.

- DEPP 76 DEPPE, M.E., and FRY, J.P. "Distributed databases : A summary of research", in Computer networks, 1, 2, North-Holland, Amsterdam, Sept. 1976.
- ESWA 76 ESWARAN, K.P., GRAY, J.N., LORIE, R.A., and TRAIGER, I.L, "The notions of consistency and predicate locks in a database system", Commun. ACM 19, 11 (Nov. 1976) pp. 624-633.
- GARC 79a GARCIA-MOLINA, H. "Performance of update algorithms for replicated data in a distributed database", Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., June 1979.
- GRAY 78 GRAY, J.N. "Notes on database operating systems", in Operating Systems : An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, pp. 393-481.
- HAMM 80 HAMMER, M.M., and SHIPMAN, D.W. "Reliability mechanisms for SDD-1 : A system for distributed databases", ACM Trans. Database Systems 5, 4 (Dec. 1980), pp. 431-466.
- HOLT 72 HOLT, R.C. "Some deadlock properties of Computer Systems", Comput. Surv. 4, 3 (Dec. 1972), pp. 179-195.
- KING 74 KING, P.F., COLLMEYER, A.J. "Database Sharing - an efficient method for supporting concurrent processes", in Proc. 1974 Nat. Computer Conf., Vol. 42, AFIPS Press, Arlington, Va., 1974.
- KOHL 81 KOHLER, W.H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", Comput. Surv. 13, 2 (June 1981), pp. 149-183.
- LAMP 76 LAMPSON, B., and STURGIS, H. "Crash recovery in a distributed data storage system," Tech. Rep, Computer Science Lab., Xerox Palo Alto Research Centre, Palo Alto, Calif., 1976.

- MENA 79 MENASCE, D.A., and HUNTZ, R.R. "Locking and deadlock detection in distributed databases," IEEE Trans. Softw. Eng. SE-5, 3 (May 1979), pp. 193-202.
- MILE 79 MILENKOVIC, M. "Synchronization of concurrent updates in redundant distributed databases", Distributed Databases - edited by C. Delobel and W. Litwin, Inria Sirius, north Holland, 1980 (original paper appeared in 1979).
- PAPA 77 PAPADIMITRIOU, C.H., BERNSTEIN, P.A., and ROTHNIE, J.B. "Some computational problems related to database concurrency control", in Proc. Conf. Theoretical Computer Science, Waterloo, Ont. Canada, Aug. 1977.
- PAPA 79 PAPADIMITRIOU, C.H. "Serializability of concurrent updates", J. ACM 26, 4 (Oct. 1979) pp. 631-653.
- ROSE 78 ROSENKRANTZ, D.J., STEARNS, R.E., and LEWIS, P.M. "System Level Concurrency control for distributed database systems", ACM Trans. Database System 3, 2 (June 1978), pp. 178-198
- ROTH 77 ROTHNIE, J.B., and GOODMAN, N. "A survey of research and development in distributed database systems", in Proc. 3rd Int. Conf. Very Large Data Bases (IEEE), Tokyo, Japan, Oct. 1977.
- ROTH 80 ROTHNIE, J.B. Jr., BERNSTEIN, P.A., FOX, S., GOODMAN, N., HAMMER, M., LANJERS, T.A., REEVE, C., SHIPMAN, D.W., and WONG, E. "Introduction to a system for distributed databases (SDDL)", ACM Trans. Database Syst. 5, 1 (March 1980), pp. 1-17.
- SCHR 80 SCHREIBER, F.A., BALDISSERA, C., CERL, S. "Distributed Database Applications : An overview", Distributed Databases - An advanced course - Cambridge University Press, Cambridge 1980.

- STEA 76 STEARNS, R.E., LEWIS, P.M. II, and
ROSENKRATZ, D.J. "Concurrency controls
for database systems", in Proc. 17th
Symp. Foundations Computer Science (IEEE),
1976, pp. 19-32.
- STON 79 STONEBRAKER, M. "Concurrency control and
consistency of multiple copies of data in
distributed INGRES, IEEE Trans. Softw. Eng.
SE-5, 3 (May 1979), pp. 188-194.
- THOM 79 THOMAS, R.H. "A solution to the concurrency
control problem for multiple copy databases",
in Proc. 1978 COMPCON Conf. (IEEE), New York.
- ULLM 84 ULLMAN, J.D. - Principles of Database
Systems (2nd Edition) - Galgotia Publications.

