Chapter 1
Introduction

1.1 Introduction

Database is the collection of related data that represents some aspect of real world. In another term, database is the source of data with which the interested user can interact to find some useful information in a systematic way. As database can be of any size and of varying complexity, database and database system have become an important aspect of everyday life in modern society. In addition, recent progress in communication and database technologies has changed the environment in which the user data is processed. Nowadays, user applications require data access from various data sources. These data sources may be located in different environments and distributed over the network. Furthermore, a database offers many advantages compared to a simple file system with regard to speed, accuracy, and accessibility such as: shared access, minimal redundancy, data consistency, data integrity, and controlled access. All of these aspects are enforced by a database management system. "A distributed database (DDB) is a collection of multiple, *logically interrelated* databases *distributed over a computer network*. A DDBMS is the software system that permits the management of DDBs and makes the distribution transparent to the user." A Distributed System is a number of autonomous computers communicating over a Network with software for integrated tasks. In short a *distributed database* is a collection of databases that can be stored at different computer network sites. Each database may involve different database management systems and different architectures that distribute the execution of transactions [7]. The objective of a distributed database management system (DDBMS) is to control the management of a distributed database (DDB) in such a way that it appears to the user as a centralized database [18].

1.2 Objective of the Study

The specific objective of this study was to develop and to analyze a lock based concurrency control algorithm (Two-Phase) in distributed database system. The general objectives of this study were stated as:

1. To analyze the distributed database system architecture,
2. To differentiate the distributed database from other database systems.
3. To find out the necessity of different transaction management model for distributed database, and
4. To find out the deadlock problem in distributed database.

1.3 Significance and Limitations of the Study

This study focuses on the concurrency control (Two-phase locking algorithm) for distributed database system. This also introduces the basic concepts associated with the distributed database environment and its architecture. Every research work has to face some limitations. This study was completed within a confined time with limited resources. However, the researcher had tried to make every possibility to carry out the study works more accurately as far as possible rather than just being perfunctory.

1.4 Thesis Structure

This section introduced some idea about the distributed database system and motivation behind this. This section also presented some aspects of this study.

Chapter 2 “Components of Distributed Database” includes an introduction to the distributed database, a discussion on the distinguishing features of the system, and describes different components and distributed database system architecture including transaction manager and scheduler.

Chapter 3 “Transaction Processing” describes different properties of transaction and how transaction takes place in distributed database management system.

Chapter 4 “Concurrency Control Problem” describes the major concurrency control problem: dirty read problem, fuzzy read problem, lost update problem, inconsistent retrievals problem and phantom problem. These problems are generally arises in database management system due to concurrent access in database system. Concurrency control algorithms must deals with these problems to ensure database consistency. Moreover, this chapter describes non-recoverability and cascading abort as a concurrency control problems.

Chapter 5 “Serializability” describes Serializability theory in detail, describes Serial schedule, Serializability test and Recoverability. Serializability is a major correctness criteria of concurrent control problems. Execution history of each concurrency control algorithm must be serializable to ensure the correctness of concurrency control algorithm.

Chapter 6 “Concurrency Control via Locking” describes the locking principle and various concurrency control algorithms in locking protocol. 2PL is a major concurrency control algorithm in locking, it describes in detail including Transaction manager and Lock manager’s algorithm. This chapter also describes deadlock management and detection technique.

Chapter 7 “Implementation and Testing” has described a complete organization of the program to implement the model presented in the sixth chapter. Sample testing and output analysis has been presented in this chapter.

Chapter 8 “Conclusion and Further Recommendations” has presented the concluding remarks of this study and future work on the concurrency control in distributed database management system.

1.5. Problem definition

Distributed concurrency control, by contrast, is in a state of extreme turbulence. More than 20 concurrency control algorithms have been purposed for DDBMSs, and several have been, or are being implemented. These algorithms are complex, hard to understand and difficult to prove correct. We will introduce a standard terminology for describing DDBMS concurrency control algorithms and a standard model for the DDBMS environment. For analysis purpose we decompose the concurrency control problem into two major sub problems called read-write and write-write synchronization. Different researchers have published papers in the field of distributed database system in its transaction processing and concurrency control system. In the case of transaction of data, different problems occurs such as the unrepeatable read problem, the lost update problem, the temporary update problem, the incorrect summary problems etc. To solve such types of problems in data transaction, different researchers used different algorithms such as two phase locking techniques and timestamp based concurrency control algorithm. The main purpose of concurrency control is to enforce isolation among conflicting transaction, to preserve database consistency through consistency preserving execution of transactions, to resolve read-write and write-write conflicts.

We are able to show the different algorithms used in concurrency control of distributed database system and its implementation in JAVA.

1.6 Literature Survey

In this phase, we shall study various research papers about the distributed database and we shall going to study other different texts and research papers in the transaction processing and concurrency control algorithms which helps us to complete our research work.

Chapter 2

Components of Distributed Database

2.1 Introduction

The database system components are the foundation for the study of concurrency control in database management system. This chapter briefly describes the distributed database system and its components providing the foundation for the study of concurrency control in database management system.

In a distributed database system, the database is stored on several computers. The computers on a distributed database system communicate with one another through various communication media. The computers on a distributed database system are referred to by a number of different names, such as sites or nodes. In distributed database system there are mainly two types of transactions: local transactions and global transactions. A local transaction is one that accesses data only from sites where the transaction was initiated. A global transaction, on the other hand, is one that either accesses data in several different sites.

There are several reasons for building distributed database systems, including sharing of data, autonomy and availability [42, 47].

1. **Sharing data**. The major advantage in building a distributed database system is the provision of an environment where users at one site may be able to access the data residing at other sites. For example, in a distributed banking system, where each branch stores data related to that branch, it is possible for a user in one branch to access data in another branch.

2. **Autonomy**. The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored

locally. In centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site. Depending on the design of the distributed database system, each administrator may have a different degree of local autonomy. The possibility of local autonomy is often a major advantage of distributed databases.

3. **Availability**. If one site falls in a distributed system, the remaining sites may be able to continue operating. In particular, if data items are replicated in several sites, a transaction needing a particular data item may find that item in any of several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

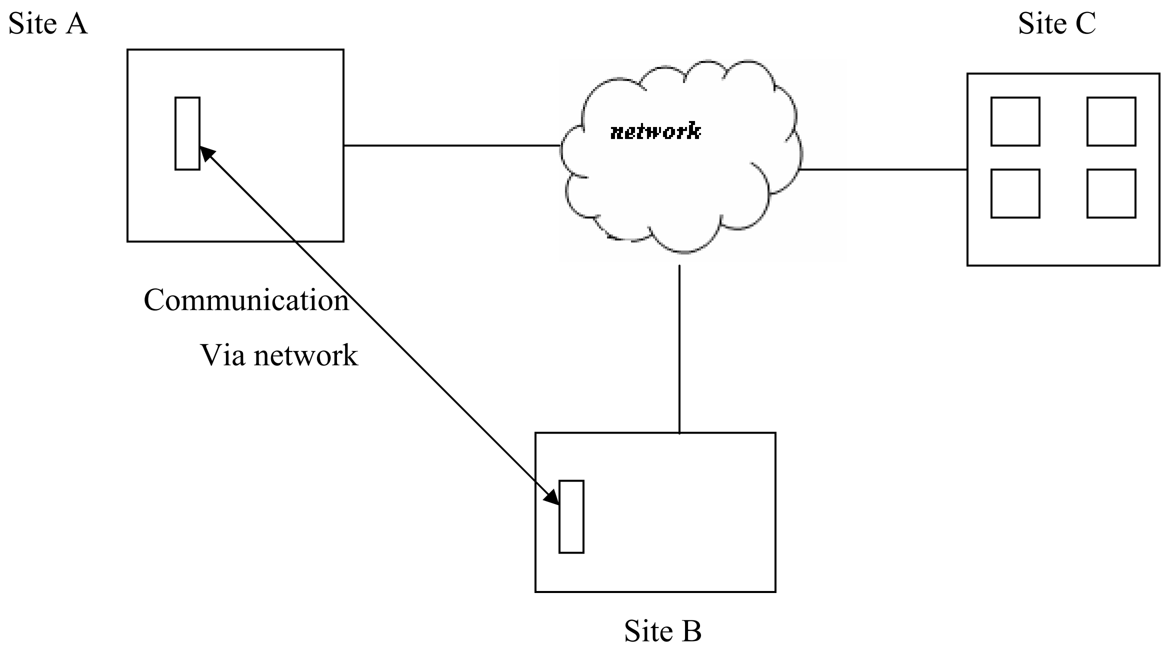


Figure 2.1 A distributed system.

2.2 Homogeneous and Heterogeneous Database

In a homogeneous distributed database [20], all sites have identical database management system software, are aware of one another, and agree to cooperate in processing user's requests. In such a system, local sites surrender a portion of heir autonomy in terms of their right to change schemas or datable management system software. The software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.

In a heterogeneous distributed database [17], different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. In a heterogeneous distributed system, individual systems can have different architectures and query languages.

2.3 Distributed Database System Architecture

In general, database system consists four components: Transaction Manager(TM), Scheduler, Recovery Manager (RM) and Transaction Coordinator. Transaction Manager is responsible to perform any required preprocessing for database and transaction operations that receives from transaction. Scheduler is major component for concurrency control. It is responsible to control the relative order of database and transaction operations to execute. Recovery Manager (RM) is major component for recovery from failures. It is responsible [7, 17] to commit and abort the transaction. A transaction coordinator is responsible for coordinating the execution of all the transactions initiated at that site.

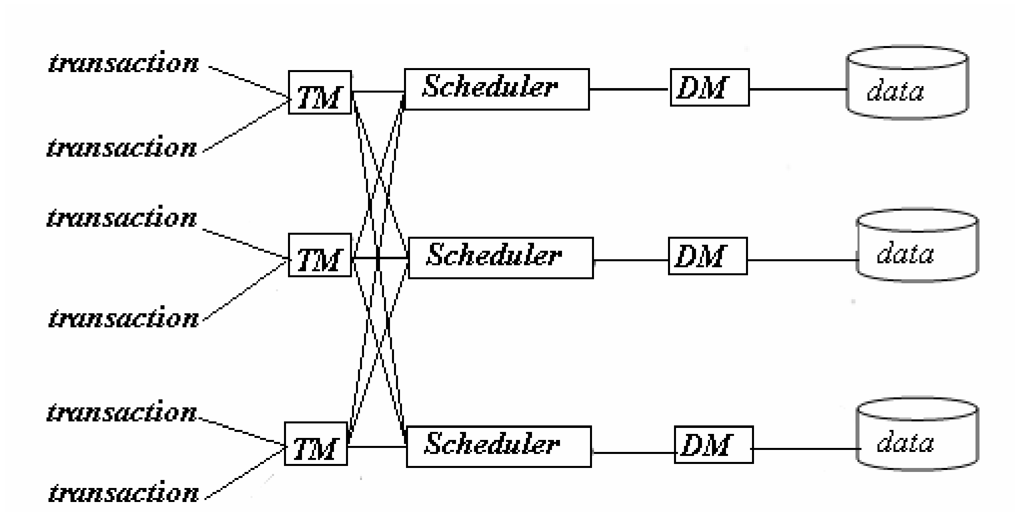


Figure2.2 Distributed Database System Architecture

2.3.1 The Scheduler and Data Manager

The scheduler is primary database system component for concurrency control. The scheduler controls the order in which DM's process Reads and Writes. When a scheduler receives a Read or Write operation, it can either output the operation right away (usually to a DM, sometimes to another scheduler), delay the operation by holding it for later action, or reject the operation. A rejection causes the system to abort the transaction that issued the operation: every Write processed on behalf of the transaction is undone (restoring the old value of the data item), and every transaction that read a value written by the aborted transaction is also aborted. This phenomenon of one abort triggering other aborts is called cascading aborts. (It is usually avoided in commercial DBS's by not allowing a transaction to read another transaction's output until the DBS is certain that the latter transaction will not abort.) In fact, scheduler is a program, based on concurrency control algorithms for serializable execution of database and transaction operations [36].

There are three basic actions scheduler performs once scheduler receives database and transaction’s operations from transaction.

Execute: Scheduler pass transaction's operation to Data Manager (DM) to execute. When DM finishes execution of passed operation it informs scheduler. Moreover, if operation is read, it reads data value from database and it relays back to transaction.

Reject: Scheduler may reject to process the operation which causes transaction to be aborted. Abort can be issued by transaction or TM.

Delay: Scheduler may delays operation placing in a queue. Later scheduler can either execute or reject it.

The three actions of scheduler are preliminary to control the order of execution of database and transaction's operations. When it receives an operation from the transaction, it usually tries to pass it on the DM. if it is unable to execute without producing non-serializable execution, either it delays or reject it. If scheduler finds possibility to correctly process operation in future it simply delays the operation.

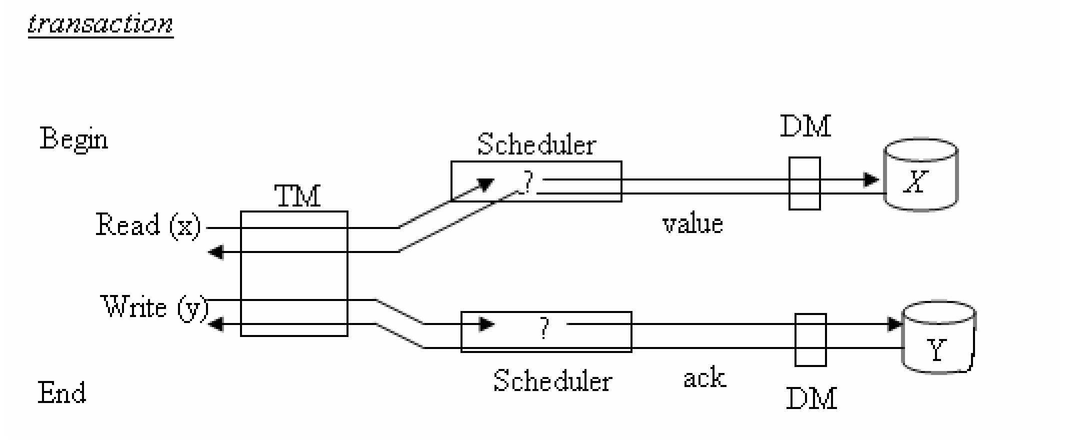


Figure: 2.3 Processing Operations

The DM executes each Read and Write it receives. For Read, the DM looks in its local database and returns the requested value. For Write, the DM modifies its local database and returns an acknowledgment. The DM sends the returned value or acknowledgment to the scheduler, which relays it back to the TM, which relays it back to the transaction. DM's do not necessarily execute operations first-come-first-served. If a DM receives a Read(x) and a Write(x) at about the same time, the DM is free to execute these operations in either order. If the order matters (as it probably does in this case), it is the scheduler's responsibility to enforce the order. This is done by using a handshaking communication discipline between schedulers and DM's **(**Figure 2.4**):** if the scheduler wants Read(x) to be executed before Write(x), it sends Read(x) to the DM, waits for the DM's response, and then sends Write(x). Thus the scheduler doesn't even send Write(x) to the DM until it knows Read(x) was executed. Of course, when the execution order doesn't matter, the scheduler can send operations without the handshake. Handshaking [8] is also used between other modules when execution order is important. To execute Read(x) on behalf of transaction 1 followed by Write(x) on behalf of transaction 2.

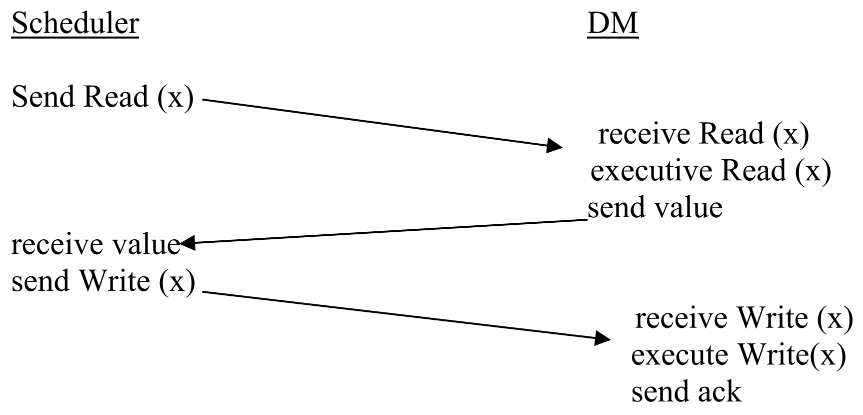


Figure 2.4 handshaking

2.3.2 Transaction Manager

The major function of transaction manager is to establish the communication between user transaction and database. That is, transaction interacts with the database through a transaction manager (TM). The TM receives database and transactions operations issued by transactions and forwards them to the scheduler. If transaction is aborted transaction manager (TM) is responsible to resubmit the transaction to scheduler. In distributed database system environment TM is more responsible, it has to decide in which site transaction operation has to send for scheduler. TM supervises transactions. Each transaction executed in the distributed database management system is supervised by a single TM, meaning that thee transaction issues all of its database operations to that TM.

Any distributed computation that is needed to execute the transaction is managed by TM. Each site of a DDBS runs one or more of the following software modules (Figures2.2 and 2.3): a transaction manager (TM), a data manager (DM) or a scheduler. Transactions talk to TM's; TM's talk to schedulers; schedulers talk among themselves and also talk to DM's: and DM's manage the data.

Each transaction issues all of its Reads and Writes to a single TM. A transaction also issues a Begin operation to its TM when it starts executing and an End when it's finished. The TM forwards each Read and Write to a scheduler. (Which scheduler depends on the concurrency control algorithm; usually, the scheduler is at the same site as the data being read or written. In some algorithms, Begins and Ends are also sent to scheduler).

The transaction manager [25] manages the execution of those transactions that access data stored in local site. Each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites). Each transaction manager is also responsible for 1) maintaining a log for recovery purposes 2) participating is an appropriate concurrency control scheme to coordinate the concurrent execution of the transactions executing at that site.

2.3.3 Transaction Coordinator

The transaction coordinator coordinates the execution of the various transactions (both local and global) initiated at that site. The transaction coordinator [19] subsystem is not needed in the centralized environment, since a transaction access data at only a single site. For each such transaction, the coordinator is responsible for

1. Starting the execution of the transaction.
2. Breaking the transaction into a number of sub transactions and distributing these sub transactions to the appropriate sites for execution.
3. Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

Chapter 3

Transaction Processing

3.1 Motivation

Concurrency is a mandatory property of a database system it must allow by the database system. In concurrent environment, read and write operation of one database user may interface with other. Due to interference only some read/write operations of database user may execute rest of read/write operations could not be executed since database system assumes each read/write operation as individual and independent task. if all read and write operations are issued by database user are really independent in nature, partial execution of read/write operations does not create big problem. But in reality, each database read or write operations really represent a complete task of database user. In such situation, it may lead inconsistency problem [9, 35]. This really demands encapsulation of set of database operations which can perform a complete task. In fact, transaction is initiated with this concept. It isolates set of database operations providing set of operations as a single unit. If any one of the operations that exists in set of database operation could not execute either because of concurrent transaction interface or because of failure, database system ignores set of all operations that exist. This helps to ensure consistency of database in concurrent environment. That is the major motivation of transaction is to ensure consistency allowing concurrent execution.

3.2 Transaction

Users interact with the DDBS by executing programs called transactions. A transaction only interacts with the outside world by issuing Reads and Writes to the DDBS or by doing terminal I/O. A transaction [25] is a unit of program consisting set of database operations whose execution may change the database state. Transaction can also define as a collection of actions that make consistent transformations preserving database consistency. To ensure consistency of database before and after execution of transaction, it needs to be atomic. Read, Write, Commit and Abort are major database operations that exist in transaction.

Formal Definition of Transaction

We can define a transaction T_i as a partial ordering over its operation and the termination condition. A partial order P = $\{\sum, \prec\}$ defines an ordering among elements of $\sum$ (called domain) according to an transitive binary relation $\prec$ defined over $\sum$. $\sum$ consists of the operations and termination condition of a transaction, whereas $\prec$ indicates the execution order of these operations. Formally, then, a transaction Ti as a partial order Ti= $\{\sum_i, \prec_j\}$, where,

1. $\sum_i = OS_i \cup \{N_i\}$
2. For any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = \{R(x) or W(x)\}$ and $O_{ik} = W(x)$ for any data item x, then either $O_{ij} \prec_i O_{ik} or O_{ik} \prec_i O_{ij}$
3. $O_{ij} \in OS_i, O_{ij} \prec_i N_i$

Where $O_{ij}(x)$ denote some operation O_j of transaction T_i that operates on a database entry x. $O_{ij} \in$ {read, write}. Operations are assumed to be atomic. OS_i denote the set of all operations in T_i (i.e., OS_i = $(\cup_{ij} O_{ij})$. And N_i denote the termination condition for T_i, where $N_i \in$ {abort, commit}.

The first condition formally defines the domain as the set of read and write operations that make up the transaction, plus the termination condition, which may be either commit or abort. The second condition specifies the ordering relation between the conflicting read and write operations of the transaction, while the final condition indicates that the termination condition always follows all other operations.

Example 3.2.1

Consider the simple transaction T that consists of following steps:

Read (x)

Read (y)

x ← x + y

Write (x)

Commit

The specification of this transaction according to the formal notation is:

$\sum$ = {R(x), R(y), W(x), C}

$\prec$ = {(R(x), W(x)), (R(y), W(x)), (W(x), (CT)), (R(x), C), (R(y), C)}

Where (O_i, O_j) as an element of the $\prec$ relation indicates that $O_i \prec O_j$.

3.3 Distributed Transaction- Processing Model

Transaction processing in a distributed environment [7] differs from that in a centralized one in two areas: handling private workspace and implementing two phase commit. In a centralized DBMS we assumed that (1) private workspaces were part of the TM, and (2) data could freely move between a transaction and its workspace, and between a workspace and the DM. These assumptions are not appropriate in a DDBMS because TMs and DMs may run at different sites and the movement of data between a TM and a DM can be expensive. To reduce this cost, many DDBMSs employ *query optimization procedures* which regulate the flow of data. The problem of atomic commitment is aggravated in a DDBMS by the possibility of one site failing while the rest of the system continues to operate. Suppose T is updating x, y, z stored at DMx, DMy, DMz, and suppose T's TM fails after issuing dm-write(x), but before issuing the dm-writes for y and z. At this point the database is incorrect. In a centralized DBMS this phenomenon is not harmful because no transaction can access the database until the TM recovers from the failure. However, in a DDBMS, other TMs remain operational and can access the incorrect database. To avoid this problem [7, 18], prewrite commands must be modified slightly. In addition to specifying data items to be copied onto secure storage, pre writes also specify which other DMs are involved in the commitment activity. Then if the TM fails during the second phase of two-phase commit, the DMs whose dm-writes were not issued can recognize the situation and consult the other DMs involved in the commitment. If any DM received a dm-write, the remaining ones act as if they had also received the command.

As in a centralized DBMS, a transaction T accesses the system by issuing BEGIN, READ, WRITE, and END operations. In a DDBMS these are processed as follows.

BEGIN: The TM creates a private workspace for T. We leave the location and organization of this workspace unspecified.

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 selects some stored copy of X, say x_i, and issues dim-read (x_i) to the DM at which x, is stored. The DM responds by retrieving the stored value of x_i from the database, placing it in the private workspace. The TM returns this value to T.

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

END: Two-phase commit begins. For each X updated by T, and for each stored copy x_i of X, the TM issues a prewrite (x_i) to the DM that stores xi. The DM responds by copying the value of X from T's private workspace onto secure storage internal to the DM. After all prewrites are processed, the TM issues dm-writes for all copies of all logical data items updated by T. A DM responds to dm-write (x_i) by copying the value of x_i from secure storage into the stored database. After all dm-writes are installed, T's execution is finished.

3.4 Properties of Transaction

The definition of transaction tells states of transaction and its actions are not visible to other transactions or database users until transaction terminates. That is, partial changes made by transaction are not visible outside this transaction. Only when transaction terminates, database users notified its success or failure and changes made by transaction are made visible. We already discuss that these are the foundation for concurrency control. To achieve these characteristics, transaction should have atomicity, consistency, isolation, and durability properties, called ACID properties [47, 15, 25, 27] of transaction.

3.4.1 Atomicity

Atomicity refers to the fact that a transaction is treated as a unit of operation. Therefore, either all the transaction's actions are completed, or none of them are. This is also known as the "all-or-nothing property". We have just extended the concept of atomicity from individual operations to the entire transaction. Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure. There are, of course, two possible courses of actions: it can either be terminated by completing the remaining actions, or it can be terminated by undoing all the actions that have already been executed. A transaction itself may fail due to input data errors, deadlocks, or others factors. Maintaining transaction atomicity in the presence of this type of failure is commonly called the transaction recovery. Let us consider a task, which is responsible to transfer funds from account A to B. Assume that failure occurs (power failure or hardware failure or software error) immediately account A is updated but before update perform in account B. Definitely, such incomplete transaction leads database in inconsistent state such incomplete execution of transaction's effect should wipe out. Transaction's atomicity property does not allow violating such integrity [9, 35]. Transaction Manager (TM) is responsible for ensuring atomicity property of transaction.

3.4.2 Consistency

The consistency property [19, 20] of transaction ensures transaction should preserve consistency of database during its execution. The consistency of a transaction is simply its correctness. In other words, a transaction is a correct program that maps one consistent database state to another. A correct execution of the transaction must take the database from one consistent state to another.

3.4.3 Isolation

Isolation is the property of transactions which requires each transaction to be consistent database at all times. In other word, an executing transaction cannot reveal its result to other concurrent transactions before its commitment. There are a number of reasons for insisting on isolation. One has to do with maintaining the inter consistency of transactions. If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value. A transaction should not make its updates visible to other transactions until it is committed. The database system component scheduler is responsible for ensuring isolation property of transaction.

Example 3.4.3.1

T_1	T_2
Read (x)	Read (x)
$x \leftarrow x + 1$	$x \leftarrow x + 1$
Write (x)	Write (x)
Commit	Commit

The following is one possible sequence of execution of the actions of these transactions:

T_1: Read(x)

T_1: $x \leftarrow x + 1$

T_1: Write(x)

T_1: Commit

T_2: Read(x)

T_2: $x \leftarrow x + 1$

T_2: Write(x)

T_2: Commit

In this case there are no problems: transaction T_1 and T_2 are executed one after the other and transaction T_2 reads 51 as the value of x. If T_2 executes before T_1, T_2 reads 51 as the value of x. So, if T_1 and T_2 are executed one after other, the second transaction will read 51 as the value of x and x will have 52 as its value at the end of execution of these two transactions. However, since transactions are executing concurrently, the following execution sequence is also possible:

T_1: Read(x)

T_1: $x \leftarrow x + 1$

T_2: Read(x)

T_1: Write(x)

T_2: $x \leftarrow x + 1$

T_2: Write(x)

T_1: Commit

T_2: Commit

In this case, transaction T_2 reads 50 as the value of x. This is incorrect since T_2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T_1 and T_2 since T_2's write will overwrite T_1's write.

The problems occurs in these examples will be explain in next chapter (concurrency control problem).

3.4.4 Durability

Durability [21] refers to that property of transactions which ensures that once a transaction commits, its result are permanent and cannot erased from the database. Or, once a transaction changes the database and the changes area committed, these changes must never be lost because of subsequent failure. The recovery manager is responsible for ensuring durability property of transaction. A simple ides for ensuring durability property of transaction is to keep the log of all changes carried out before writing the effect of updated transaction to disk. The concept of log can use by TM to restore the database state during the system failure or system restart.

Chapter 4

Concurrency Control Problems

4.1 Introduction

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. Generally database system allows multiple transactions to run concurrently. An important consideration in the design of distributed systems is the concurrency control. The concurrency control is that portion of the system that is concerned with deciding what actions should be taken in response to requests by the individual processes to read and write into the database. Concurrency control is the activity of coordinating concurrent access to a database in a multi-user database management system (DBMS). Concurrency control permits users to access a database in a multi- programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The goal of concurrency control is to prevent interference among users who are simultaneously accessing a database [3, 12]. The concurrency control is concerned with avoiding deadlocks or similar occurrences and with maintaining the consistency of the database. The concurrency control has the same task whether the database is centralized or distributed. Concurrent execution of transaction in database system improves database system performance, reducing transaction waiting time to proceed. It improves resource utilization. But it may leads database inconsistent state due to interference among actions of concurrent transactions. Concurrent execution of transaction in database system leads several concurrency control problems [35] that may generally arise in concurrent execution will discuss in this chapter.

4.2 Concurrency Control Problems

Concurrency control permits users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The concurrency control problem [35] is exacerbated in a distributed DBMS (DDBMS) because (1) users may access data stored in many different computers in a distributed system, and (2) a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers.

4.2.1 Dirty Read Problem

Dirty data refer to the data items whose values have been modified by a transaction that has not yet committed. Consider the case where transaction T_1 modifies a data item value, which is then read by another transaction T_2 before T_1 performs a Commit or Abort. In case T_1 aborts, T_2 has read value which never exists in the database.
A precise specification of this phenomenon as follows:

……, $W_1(x)$, ……, $R_2(x)$, ……., C_1(or A_1),…..,C_2(or A_2)
Or
……, $W_1(x)$, ……., $R_2(x)$, ……., C_2(or A_2),…..,C_1(or A_1)

Let us examine a concurrent schedule that demonstrates a possible dirty read problem.

T_1	T_2
$Write_1(x)$	
	$Read_2(x)$
	$Write_2(y)$
$Abort_1$	

Table 4.1 Schedule illustrating dirty read problem.

Since T_2 read x (dirty read) that was already written by T_1 but not committed yet T_1 aborts also cause T_2 to be abort, changes made by T_2 is never committed. Dirty read problem in concurrent execution occur if transaction T reads uncommitted transaction and subsequently aborts before T's commit [9]. Similar case is shown in above schedule.

4.2.2 Lost Update Problem

If two or more transaction modifies data item x at a time then lost update problem occurs, update made by one transaction may overwrite by other transaction update. If T_1 updates x but not committed yet, before T_1 commit, if another transaction T_2 update x and eventually T_2 commits before T_1 then update made by T_1 is lost [35]. Let us consider a schedule which demonstrates lost update problem.

T_1	T_2
$Read_1(x)$	
	$Read_2(x)$
$Write_1(x)$	
	$Write_2(y)$
	$Commit_2$

Table 4.2 Schedule illustrating Lost Update problem.

Initially T_1 and T_2 read same data value of x. when T_2 commits, T_1's write to x is overwritten by T_2's write x.

Anomaly: Lost Updates

Suppose customers simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere (Figure 4.1). The two ATMs handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the balances back into the database. The effect is incorrect: although two customers deposited money, the database only reflects one activity; the other deposit is lost by system.

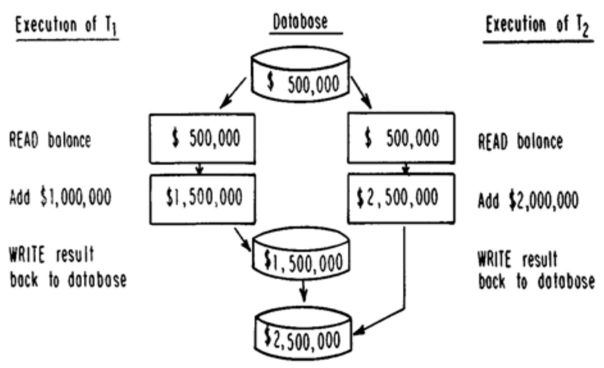


Figure: 4.1 Lost Update Problem [7].

4.2.3 Non-Repeatable (Fuzzy) Read Problem

Suppose a transaction T_1 reads a data item value. Another transaction T_2 then modifies or deletes that data item and commits. If T_1 then attempts to reread the data item, it either reads a different value or it cannot find the data item at all; thus two reads within the same transaction T_1 returns different results.

A precise specification of this phenomenon is as follows:

……, $R_1(x)$, ……, $W_2(x)$, ……., C_1(or A_1),…..,C_2(or A_2)

Or

……, $R_1(x)$, ……., $W_2(x)$, ……., C_2(or A2),…..,C_1(or A_1)

Let us examine a concurrent schedule that demonstrates a possible fuzzy read problem.

T_1	T_2
$Read_1(x)$	
	$Write_2(x)$
	$Commit_2$
$Read_1(y)$	
$Write_1(x + y \rightarrow z)$	
$Commit_1$	

Table 4.3 Schedule illustrating Fuzzy read problem.

4.2.4 Phantom Problem

Most of the databases are dynamic; meaning is that there are no fixed numbers of records in which we always perform query to update and to retrieve required data. In normal, we need to add, remove or moved data within database. Such database is called dynamic database [39, 45]. In dynamic database phantom problem may arise. When T_1 does a search with a predicate and T_2 inserts new tuples that satisfy the predicate. The predicate specification of this phenomenon is (where P is the search predicate).

……, $R_1(P)$, ……, $W_2(y \text{ in } P)$, ……., $C_1(\text{or } A_1)$,…..,$C_2(\text{or } A_2)$
Or
……, $R_1(P)$, ……., $W_2(y \text{ in } P)$, ……., $C_2(\text{or } A_2)$,…..,$C_1(\text{or } A_1)$

Suppose T_1 is responsible to read data values of x and y then need to store their sum in z and T_2 is responsible to delete the data item x. assume possible concurrent schedule with T_1 and T_2 as below.

T_1	T_2
$Read_1(x)$	
	$Delete_2(x)$
	$Commit_2$
$Read_1(y)$	
$Write_1(x + y \rightarrow z)$	
$Commit_1$	

Table 4.4 Schedule illustrating Phantom problem.

Initially T_1 reads data value of x and keep it to add with data value of y but immediately T_2 deletes data item x before T_1 store sum of x and y to z. when T_1 commits T_1 stores sum of x and y even data item x is no longer exist in database.

4.2.5 Inconsistent Retrievals

Some problems that occur when concurrent execution is uncontrolled. Suppose two transactions T_1 and T_2 are two execute simultaneously.

T_1	T_2

	Sum=0
	Read(A)
	Sum= Sum+A

Read(X)	
X= X-N	
Write(X)	

	Read(X)
	Sum= Sum+X
	Read(Y)
	Sum= Sum+Y

Read(Y)
Y=Y+N
Write(Y)	

Table 4.5 Schedule illustrating Inconsistent Retrievals

T_2 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Anomaly: Inconsistent Retrievals [7].

Suppose two customers simultaneously execute the following transactions.

Customer 1: Move $1,000,000 from savings account to its checking account.

Customer 2: Print total balance in savings and checking.

The first transaction might read the savings account balance, subtract \$1,000,000, and store the result back in the database. Then the second transaction might read the savings and checking accounts balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding \$1,000,000, and finally storing the result in the database. Unlike Lost Update Anomaly, the final values placed into the database by this execution are correct. Still, the execution is incorrect because the balance printed by Customer 2 is \$1,000,000 short.

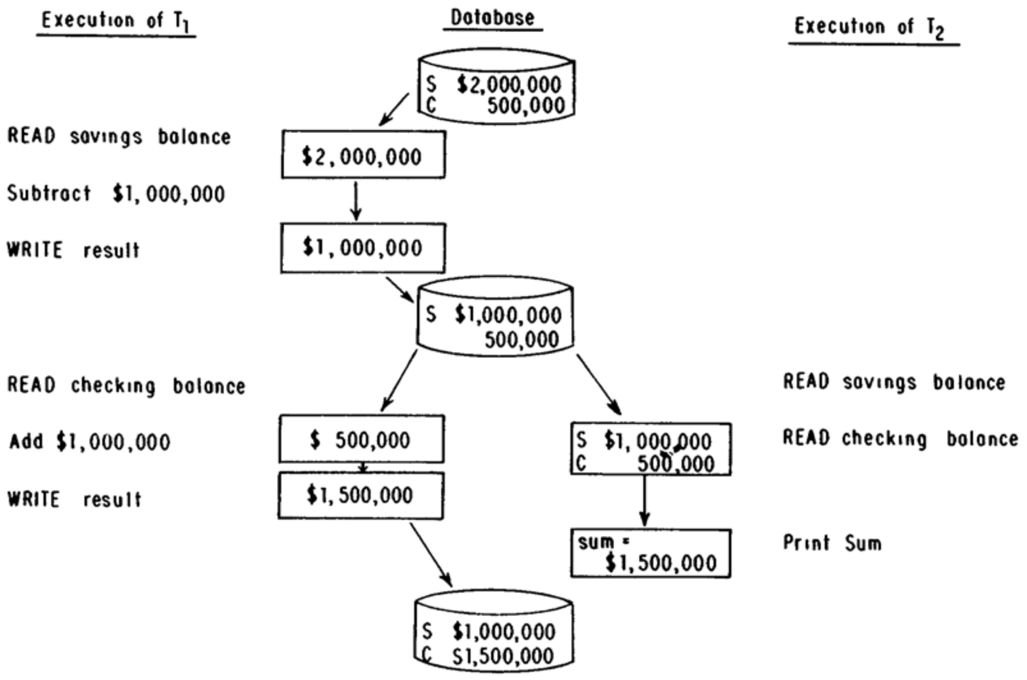


Figure 4.2 Inconsistent Retrievals [7]

4.3 Non Recoverability and Cascading Aborts as a Concurrency Control Problem

When transaction T aborts, database system must undo its effects for each data item updated by T. That is, database system need to rollback T's effect from database during abort of T. There are two possible effect of transaction T. T may effects on data value written in the database or it may also effects on other transaction. In both case, aborted transaction's effects should undo from database. If aborted transaction may trigger further abortion, it is known as cascading abort.

Let us consider a schedule which illustrates cascading abort. Assume x and y are data items having initial data value 1 for both x and y.

T_1	T_2
$Write_1(x,2)$	
	$Read_2(x)$
	$Write_2(y,3)$
$Abort_1$	

Table 4.6 Schedule illustrate Cascading Abort

In the above schedule, when transaction T_1 aborts database system must restore update made by T_1. That is, database system must undo $Write_1(x, 2)$ restoring x=1. Restoring the update made by T_1 is not sufficient since T_2 reads value of x written by T_1, T_2 also need to abort. That is, database system need to undo $Write_2$ (y, 3) restoring y=1. Even cascading abort maintain consistency of database by aborting series of transactions, it is in fact a concurrency control problem. Cascading abort is really unpleasant. This can be consider as a concurrency control problem because it required significant bookkeeping to track which transactions reads from which others, single transactions abortion force to abort one or more other transactions; which is very expensive.

Cascading abort is not always possible. Durability property of transaction tells once a transaction is committed, the database system must guarantee it could not abort. There would be a situation that where cascading abort required but not possible. This usually happen if transaction T_j reads changes made by other transaction T_i and T_i aborts after T_j's commit. Let's examine such situation by the following schedule.

T_1	T_2
$Write_1(x,2)$	
	$Read_2(x)$
	$Write_2(y,3)$
	$Commit_2$
$Abort_1$	

Table 4.7 Schedule illustrate non recoverable schedule

Here, once transaction T_1 aborts, T_2 need be aborted but it violets durability property of transaction. T_2 is already committed before T_1 aborted. So here, cascading abort is not possible. Here schedule demands cascading abort but it is not possible, such schedule called non recoverable schedule. Non-recoverability is in fact concurrency problem and recoverability is required property for concurrency control. Non-recoverable execution is more danger than cascading abort. Cascading abort is expensive but it does not violate transaction property (durability property o transaction).

Formally, recoverable schedule is defined as follows.

Suppose transaction T_j reads x that was written by transaction T_i in the execution then schedule S is called recoverable if it follows following conditions.

1. T_j reads x after T_i has written into it.
2. T_i does not abort before T_j reads x and
3. Every transaction (if any) that write x between T_i writes x and T_j reads x, aborts before T_j read x.

It indicates that, for recoverable execution if T_j reads from T_i then T_j must follow T_i's commit. An execution is recoverable if database system always able to reverse the effects of aborted transaction on other transaction [39]. Recoverability is required to ensure aborting transaction does not change the semantics of committed transaction's operations.

4.4 Avoiding Cascading Aborts and Ensuring Recoverability

We already discuss that cascading abort and non-recoverable execution are concurrency problems [1, 35], it should avoid during concurrent execution.

Cascading aborts can avoid if database ensures that every transaction read only those data values that were written by committed transactions. To achieve cascadelessness, database system need to delay each Read(x) until transaction that has previously issued a Write(x, val) have either aborted or committed. Avoiding cascading abort also ensures recoverability but enforcing recoverability does not remove the possibility of cascading aborts. Let us reexamine the schedule define in table.

T_1	T_2
$Write_1(x,2)$	
	$Read_2(x)$
	$Write_2(y,3)$
	$Commit_2$
$Abort_1$	

Table 4.8 Schedule illustrate Cascading Aborts Schedule

Here, this schedule is not cascadelesness and not recoverable. To achieve cascadelessness $Read_2(x)$ must wait till T_1's abort. And definitely it ensures recoverability as well as cascadelessness. In the above schedule if T_1 aborts just before T_2's commits, then schedule becomes recoverable but it does not avoids cascading aborts, abortion of Ti lead T_2 to abort.

4.5 Strict Execution

From the practical point of view, avoiding cascading aborts is not always enough [39]. A further restriction on execution is often desirable. The cascadelessness schedule only enforces transaction could not read data item x that was already written by uncommitted transaction. But it does not enforce transaction could not write x that was already written by uncommitted transaction. Lets examine cascadelessness schedule how it can lead problem in concurrent execution.

T_1	T_2
Write$_1$(x,2)	
	Write$_2$(y,3)
Abort$_1$	

Table 4.9 Cascadelessnes Schedule

Here, T_2 didn't read x that already written by T_1 so schedule is cascadelessness. According to the definition of cascadelessness schedule, it needs not to enforce T_2 to abort but T_2's write may dependent to T_1's write. If so such cascadlessness schedule may cause problem. Let's look the scenario more precisely, assume that initial value of x is say 50. Transaction T_1 is responsible to add 20 in x and transaction T_2 is responsible to add say 5% of current value of x then T_2's write becomes logically invalid when T_1 aborts but T_1 does not to enforce T_2 abort.

The strict execution is serious about such problem. Strict execution delays T_2's write to x until T_1 abort or commit. That is, strict execution restricts both reads and writes to x if x is already written by T_1 until T_1 is either committed or aborted. Strict execution ensures [38] both cascadelessness and recoverability.

Chapter 5

Serializability

5.1 Introduction

The concurrency control problems section we discuss different concurrency control problems that may arise in distributed database system. We also experience that these problems arise due to interference among conflict operations of concurrent transactions. Interference may cause improper order of conflicting operations [35, 30]. Here the requirement is to control the execution order of conflicting operations so that interference problem is solved. One way to avoid interference problem is not allow transaction to be interleaved at all. Execution in which no two transactions are interleaved called serial. More precisely, an execution is serial if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of other. Serial execution is always correct on the assumption that each individual transaction is correct and the transactions hat execute serially cannot interface with each other. If database system only allows serial execution, it may make poor use of its resources so it may not efficient. For only simple system, serial execution may appropriate to avoid interference. So the concept of Serializability is required for the solution of interferences problems in concurrent environment. Serializability controls the execution order of conflicting operations and ensures correctness of concurrent execution. Serializability theory can be considered as a mathematical tool to examine whether or not a scheduler works correctly. Serializability theory [7, 8, 39] tells every concurrent execution is correct if execution history of concurrent execution is serializable. Serializable execution is a major principle of Serializability theory which we will discuss in detail.

In databases and transaction processing, schedule (transaction history) is serializable, has the Serializability property, if its outcome (the resulting database state, the values of the database's data) is equal to the outcome of its transactions executed sequentially without overlapping. Transactions are normally executed concurrently (they overlap), because it is the most efficient way. Serializability is considered the highest level of isolation between transactions, and plays an essential role in concurrency control.

Serializability is the major criterion for the correctness of concurrent transactions executions. As such it is supported in all general purpose database systems. The rationale behind it is the following:

If each transaction is correct by itself, then any serial execution (at any transaction order) of these transactions is correct. As a result, any execution that is equivalent (in its outcome) to a serial execution is correct.

Schedules that are not serializable are likely to generate erroneous outcomes. Well known examples are with transactions that debit and credit accounts with money. If the related schedules are not serializable, then the total sum of money may not be preserved. Money could disappear, or be generated from nowhere. This and violations of possibly needed other invariant preservations are caused by one transaction writing, and "stepping on" and erasing what has been written by another transaction before it has become permanent in the database. It does not happen if serializability is maintained.

5.2 Serializability Theory

A schedule S (also called history) [28, 43] is defined over a set of transactions T = {T_1, T_2,… , T_n} and specifies an interleaved order of execution of these transactions operations. Based on the definition of transaction, the schedule can be specified as a partial order over T. Schedule (History) specifies the order of conflicting operations that appear in concurrent execution. Operations upon data are *read* or *write* (*insert* or *modify* or *delete*). Two operations are *conflicting*, if they are of different transactions, upon the same data item, and at least one of them is *write*. The transaction of the second operation in the pair is said to be *in conflict* with the transaction of the first operation. A more general definition of conflicting operations (also for complex operations, which may consist each of several "simple" read/write operations) requires that they are non commutative (changing their order also changes their combined result). Each such operation needs to be atomic by itself (by proper system support) in order to be commutative (non conflicting) with the otherTwo operations are said to be conflict if they both operate on the same data item and at least one of them is write. Thus, Read(x) conflicts with Write(x), while Write(x) conflicts with both Read(x) and Write(x).

Operations of different transactions		Conflict	Reason
Read	Read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed.
Read	Write	Yes	Because the effect of read and write operations depends on the order of their execution.
Write	Write	Yes	Because the effect of a pair of write operations depends on the order of their execution.

Complete schedule [1, 4], which defines the execution order of all operations in its domain. We will then define a schedule as a prefix of a complete schedule. Formally, a complete schedule S_T^C defined over a set of transaction T = {T_1, T_2,……, T_n} is a partial order $S_T^C = \{ \Sigma_T , \prec_T \}$ where

1. $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$
2. $\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$
3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T, >$ either $O_{ij} \prec_T O_{kl}$, or $O_{kl} \prec_T O_{ij}$.

The first condition simply states that the domain of the schedule is the union of the domains of individual transactions. The second condition defines the ordering relation as a superset of the ordering relations of individual transactions. The final condition simply defines the execution order among conflicting operations.

Example 5.2.1

Consider the two transactions from example 3.4.3.1. They were specified as

T1	T2
Read (x)	Read (x)
$x \leftarrow x + 1$	$x \leftarrow x + 1$
Write (x)	Write (x)
Commit	Commit

A possible complete schedule S_T^C over T = $\{T_1, T_2\}$ can be written as the following partial order (where the subscripts indicate the transactions):

$$S_T^C = \{\Sigma_T, \prec_T\}$$

Where

$$\Sigma_1 = \{R_1(x), W_1(x), C_1\}$$

$$\Sigma_2 = \{R_2(x), W_2(x), C_2\}$$

Thus

$$\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

And

$\prec_T = \{(R_1, R_2), (R_1, W_1), (R_1, C_1), (R_1, W_2), (R_1, C_2), (R_2, W_1), (R_2, C_1), (R_2, W_2), (R_2, C_2), (W_1, C_1), (W_1, W_2), (W_1, C_2), (C_1, C_2), (W_2, C_2)\}$

This can be specified as a DAG

$$\begin{tikzcd}
R_1(x) \arrow[r] \arrow[d] & R_2(x) \arrow[d] \arrow[dl] \\
W_1(x) \arrow[r] \arrow[d] & W_2(x) \arrow[d] \\
C_1 & C_2
\end{tikzcd}$$

Figure 5.1 DAG Representation of a Complete Schedule

Thus S_T^C can be specified as

$$S_T^C = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$

A schedule is defined as a prefix of a complete schedule

Example 5.2.2 Consider the following three transactions:

T_1	T_2	T_3
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

A complete schedule of these transactions is:

$$\begin{tikzcd}
R_1(x) \arrow[d] & W_2(x) \arrow[l] \arrow[r] \arrow[d] & R_3(x) \arrow[d] \\
W_1(x) \arrow[d] & W_2(y) \arrow[r] \arrow[d] & R_3(y) \arrow[d] \\
C_1 & R_2(z) \arrow[r] \arrow[d] & R_3(z) \arrow[d] \\
 & C_2 & C_3
\end{tikzcd}$$

Figure 5.2 A Complete Schedule

And a schedule S (as a prefix of complete schedule) is:

$$\begin{tikzcd}
R_1(x) & W_2(x) \arrow[l] \arrow[r] \arrow[d] & R_3(x) \arrow[d] \\
 & W_2(y) \arrow[r] \arrow[d] & R_3(y) \arrow[d] \\
 & R_2(z) \arrow[r] & R_3(z)
\end{tikzcd}$$

Figure 5.3 Prefix of Complete schedule

5.3 Serial Schedule

If in a schedule S, the operations of various transactions are not interleaved that is the operations of each transaction occur consecutively, the schedule is to be serial. The serial execution of as set of transactions maintains he consistency of the database. The three transactions of Example 5.2.2. The following Schedule,

S = $\{W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}$

Is serial since all the operations of T_2 are executed before all the operations of T1 and all operations of T_1 are executed before all operations of T3. The relationship between transaction executions can be denoted as: $T_2 \prec_S T_1 \prec_S T_3$ or $T_2 \rightarrow T_1 \rightarrow T_3$.

Two Schedules S_1 and S_2, defined over the same set of transactions T, are ***equivalent*** if they have the same effect on the database. Formally, two schedules, S_1 and S_2, defined over the same operations O_{ij} and O_{kl} $(i \neq k)$, whenever $O_{ij} \prec_1 O_{kl}$, then $O_{ij} \prec_2 O_{kl}$. This is called ***conflict equivalence*** since it defines equivalence of two schedules in terms of the relative order of execution of the conflicting operations in those schedules.

Again consider the three transactions of Example 5.2.2. S' = $\{W_2(x), R_1(x), W_1(x), C_1, R_3(x), W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3\}$.
This schedule S' is ***conflict equivalent*** to schedule S.

A schedule S is said to be ***serializable*** if and only if it is conflict equivalent to a serial schedule. Serializable is also called ***conflict-based serializability*** since it is defined according to conflict equivalence. Schedule S' is serializable since it is equivalent to the serial schedule S.

The problem with the uncontrolled execution of transactions T_1 and T_2 in Example (3.4.3.1) they could generate an unserializable schedule. In distributed system the schedule of transaction execution at each site is called a local schedule. If the database is not replicated and each local schedule is serializable, their union (called a global schedule) is also serializable as long as local serialization orders are identical. It is possible that the local schedules are serializable, but the mutual consistency of the database is still compromised.

Example 5.2.3

Consider a two sites and one data item (x) that is duplicated on both sites. Further consider the following two transactions:

T_1	T_2
Read (x)	Read (x)
$x \leftarrow x + 5$	$x \leftarrow x \times 10$
Write (x)	Write (x)
Commit	Commit

Both of these transactions need to run at both sites. Consider the following two schedules that may generated locally at the two sites:

$S_1 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$

$S_2 = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\}$

Suppose both of these schedules are serializable, they are serial. Therefore, each represents a correct execution order. However, observe that they serialize T_1 and T_2 in reverse order. Assume that the value of x prior to the execution of these transactions was 1. At the end of the execution of these schedules, the value of x is 60 at site 1 and the value of x is 15 at site 2. This violates mutual consistency of the two schedules.

Chapter 6
Concurrency Control via Locking

6.1 Introduction

To preserve the consistency of database [1, 2, 11, 12], the database system must adopt some concurrency control mechanism to ensure that the modifications made by transactions are not lost. To ensure the correctness of database, database system required to enforce some mutual exclusion in concurrent execution. One way to enforce mutual exclusion is implementing locking mechanism. Locking based strategies are most commonly used for the concurrency control method for distributed database system. The main idea of locking-based concurrency control is to ensure that the data that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a “lock” with each lock unit [22, 23, 28, 29]. This lock is set by a transaction before it is accessed and is reset at the end of its use. A lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

6.2 Locking Principle

The basic idea of locking is simple. Before accessing any database item user transaction need to acquire appropriate lock on that data item such that other transactions cannot acquire lock on same data item which leads conflict. Before terminate user transactions, it is necessary to unlock all data item that was locked by terminating transactions.

We are concerned with synchronizing the conflict operations of conflict transactions, there are two types of locks (commonly called lock modes) associated with each lock unit: Shared Lock (S-Lock) and Exclusive Lock (X- Lock) which are also called Read Lock (rl) and Write Lock (wl) respectively. A transaction Ti that wants to read a data item contained in lock unit x obtains a read lock on x denoted by $rl_i(x)$. The same happens for write operations. It is commonly talk about compatibility of lock modes.

Two lock modes are compatible if two transactions which access the same data item can obtain these locks on the data item at the same time [40, 46].

If transaction T_i holds an write lock on data item x then no other transaction T_j can achieve a lock of either types (S-Lock or X- Lock) on x until and unless T_i release associated write lock on data item x. if transaction Ti holds a read lock on data item x then no other transaction T_j can achieve write lock on data item x until T_i release associated read lock on x but it can achieve read lock on x. this locking rule can be shown by lock compatibility matrix.

	rl(x)	wl(x)
rl(x)	√	×
wl(x)	×	×

√: compatible   ×: incompatible

Table: 6.1 Compatibility of Lock modes

Two locks pl_i(x) and ql_j(y) conflict if x = y, i ≠ j and if p or q or both (p and q) are write lock. Two locks p and q on different data item x and y (x ≠ y) do not conflict.

The basic principle of locking is not sufficient for ensuring the consistency of database. Stronger scheduler required to ensure database consistency.

The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions. In other words, the users do not need to specify when data needs to be locked: the distributed DBMS [49] takes care of that every time the transaction issues a read or write operation.

In locking based system, the scheduler is a lock manager (LM). The transaction manager passes to the lock manager the database operation (read and write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then cheeks if the lock unit that contains the data item is already locked. If so and if the existing lock mode is incompatible with that of the current transactions, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

Algorithm: 6.1 (Basic Lock Manager)

Algorithm Basic-LM

```
declare-var
 msg : Message
 dop : Dbop
 Op : Operation
 x : DataItem
 T : TransactionId
 pm : Dpmsg
 res : DataVal
 SOP : OpSet
begin
   repeat
       WAIT(msg)
       case of msg
         Dbop:
         begin
            Op ← dop.opn
            $x \leftarrow dop.data$
            $T \leftarrow dop.tid$
            **Case of** Op
               Begin_transaction:
               **begin**
                  send dop to the data processor
              **end**
              Read or Write:                    {requires locking}
             **begin**
                  find the lock unit *lu* such that x $\subseteq$ *lu*
                  **if** *lu* is unlocked or lock mode of *lu* is compatible with Op then
                  **begin**
                      set lock on *lu* in appropriate mode
                      send *dop* to the data processor
                  **end**
            **else**
             put d*op* on a queue for *lu*
           **end-if**
         **end**
         Abort or Commit:
         **begin**
            send *dop* to the data processor
         **end**
   **end-case**
Dpmsg:                              {acknowledgment from the data processor}
**Begin**                                        {requires locking}
  $Op \leftarrow pm.opn$
  $res \leftarrow pm.result$
  $T \leftarrow pm.tid$
   find lock unit lu such that  x $\subseteq$ *lu*  release lock on *lu* held by T
```

```
        if there are no more locks on lu and
           there are operations waiting in queue for lu then
        begin
            SOP ← first, operation from the queue
            SOP ← SOP ∪ {O|O is an operation on queue that
                        can lock lu in a compatible mode with
                        the current operations in SOP}
              Set the locks on lu on behalf of operation is SOP
              for all the operations in SOP do
                send each operation to the data processor
                end-for
              end-if
            end
        end-case
    until forever
end. {Basic-LM}
```

The basic algorithm that is given in algorithm 6.1 will not unfortunately, properly synchronize transaction executions. This is because to generate serializable schedules, the locking and releasing operations of transactions also need to be coordinated.

Example 6.2.1

Consider the following two transactions:

T_1	T_2
Read (x)	Read (x)
$x \leftarrow x + 1$	$x \leftarrow x * 2$
Write (x)	Write (x)
Read (y)	Read (y)
$y \leftarrow y - 1$	$y \leftarrow y * 2$
Write (y)	Write (y)
Commit	Commit

The following is a valid schedule that a lock manager employing the algorithm 6.1 and that may generate:
S = {$wl_1(x)$, $R_1(x)$, $W_1(x)$, $lr_1(x)$, $wl_2(x)$, $R_2(x)$, $W_2(x)$, $lr_2(x)$, $wl_2(y)$, $R_2(y)$, $W_2(y)$, $lr_2(y)$, C_2, $wl_1(y)$, $R_1(y)$, $W_1(y)$, $lr_1(y)$, C_1}

Here lr_i indicates the release of the lock on z that transaction T_i holds. Note that S is not a serializable schedule. For example, if prior to the execution of these transactions, the values of x and y are 50 and 20, respectively, one would except their values following execution to be, respectively, either 102 and 38 if T_1 executes before T_2, or 101 and 39 if T_2 executes before T_1. However, the result of executing S would give x and y the values 102 and 39. Obviously, S is not a serializable.

The problem with schedule S in Example 6.2.1 is the following.
The locking algorithm releases the locks that are held by a transaction (say Ti) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed. However, the transaction itself is locking other items (say y), after it releases its lock on x. Even though this may seem to be advantageous from the view point of increased concurrency, it permits transaction to interfere with one another, resulting in the loss of total isolation and atomicity.

6.2.1 Two-phase locking (2PL)

Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data item x, a transaction must “own” a read lock on x. before waiting into x, it must “own” a write lock on x. the ownership of locks is governed by two rules.

1. Different transactions can not simultaneously own conflicting locks and
2. Once a transaction surrenders ownership of lock, it may never obtain additional locks.

To, solve the problems that occur in above algorithm, we use the concept of 2PL. Two-phase locking rule simply states that [49, 44, 40] no transaction should request a lock after it releases one of its lock. Alternatively, a transaction should not release a lock until it is certain that it will no request another lock. 2PL algorithms execute transaction in two phases:

Growing Phase

Each transaction has a growing phase, where it obtains a locks and accesses data items.

Shrinking Phase

In shrinking phase, it releases locks.

The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction. It is well-known theorem that any schedule generate by a concurrency control algorithm that obeys the 2PL rule is serializable.

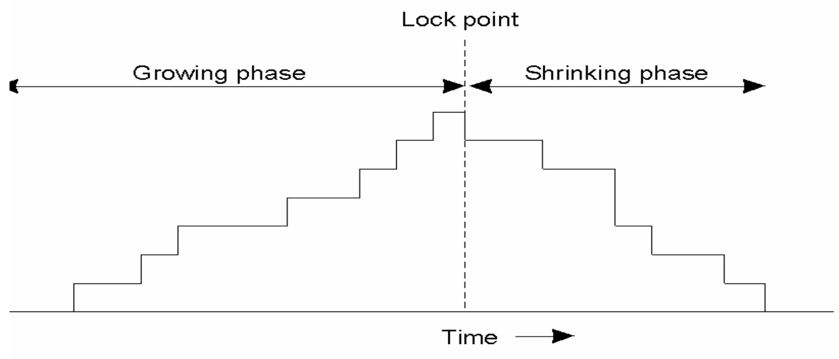


Figure 6.2 2PL Lock Graph

Figure 6.2 indicates that the lock manager releases locks as soon as access to the data item has been completed. This permits other transaction awaiting access to go ahead and lock it, there by increasing the degree of concurrency [41]. However, this difficult to implement since the lock manager has to know that the transaction has obtained al its lock and will not need to lock another data item. The lock manager also needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released.

If the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts.*

6.2.2 Strict two-phase locking

Because of these difficulties, most 2PL schedulers implement what is called strict two-phase locking [3, 8, 9], which releases all the locks together when the transaction terminates (commits or aborts).

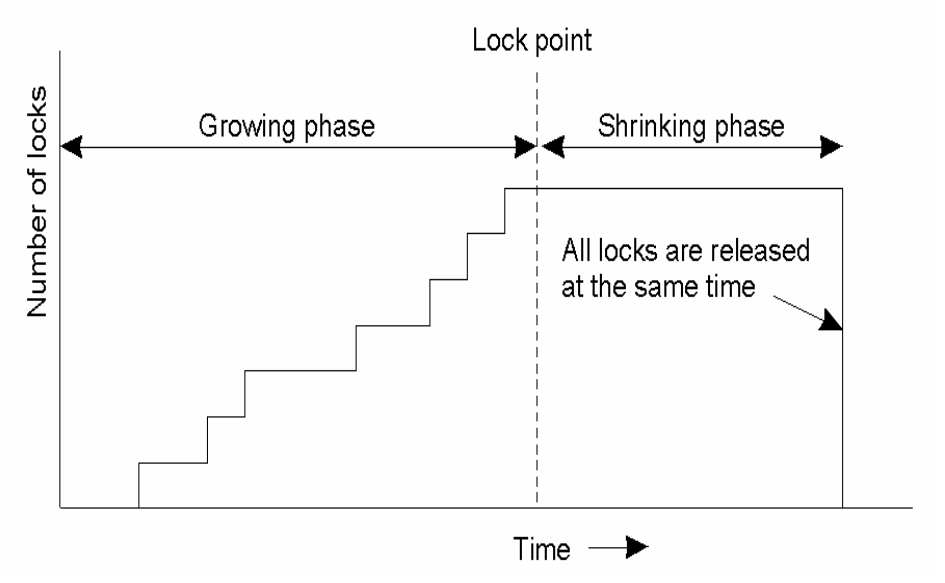


Figure 6.3 Strict 2PL Lock Graph

This is necessary to ensure that the locks are released only if the operation is a commit or abort. So, there is some modification of the algorithm 6.1.

Algorithm 6.2 (Strict Two-Phase Locking**)**

Algorithm S2PL-LM

```
declare-var
 msg : Message
 dop : Dbop
 Op : Operation
 x : DataItem
 T : TransactionId
 pm : Dpmsg
 res : DataVal
 SOP : OpSet
begin
    repeat
       WAIT(msg)
       case of msg
         Dbop:
         begin
           Op ← dop.opn
            x  ← dop.data
            T ← dop.tid
            Case of Op
               Begin_transaction:
               begin
                  send dop to the data processor
              end
              Read or Write:                        {requires locking}
             begin
                  find the lock unit lu such that x ⊆ lu
                  if lu is unlocked or lock mode of lu is compatible with Op then
```

```
                begin
                    set lock on lu in appropriate mode
                    send dop to the data processor
                end
          else
            put dop on a queue for lu
          end-if
        end
        Abort or Commit:
        begin
           send dop to the data processor
        end
    end-case
Dpmsg:
begin
  Op ← pm.opn
  res ← pm.result
  T ← pm.tid
  If Op = Abort or Op= Commit  then
  begin
     for  each lock unit lu locked by T do
     begin
        release lock on lu held by T
        if there are no more locks on lu and
           there are operations waiting in queue for lu then
        begin
           SOP ← first, operation from the queue
           SOP ← SOP ∪ {O|O} is an operation on queue that
                         can lock lu in a compatible mode with
                          the current operations in SOP}
             Set the locks on lu on behalf of operation is SOP
```

```
                 **for all** the operations in SOP **do**
                  send each operation to the data processor
                  **end-for**
                 **end-if**
               **end-for**
              **end-if**
             **end**
           **end-cae**
         **until** forever
     **end**. {S2PL-LM}
```

Algorithm 2PL-TM

```
***declare-var***
  *msg : Message*
  *Op : Operation*
  *x : DataItem*
  *T : TransactionId*
  *O : Dbop*
  *sm : Scmsg*
  *res : DataVal*
  *SOP : OpSet*
**begin**
    **repeat**
       WAIT(*msg*)
     **case of** msg
          *Dbop :*
          **begin**
               send O to the lock manager
           **end**
          *Scmsg :*                    {acknowledgement from the lock manager}
```

```
            begin
              Op ← sm.opn
              res ← sm.result
              T ← sm.tid
              Case of Op
                    Read :
                    begin
                        return res to the user application            (i.e., the transaction)
                    end
                    Write:
                    begin
                        inform user application of completion of the write
                        return res to the user application
                    end
                    Commit:
                    begin
                        destroy T's workspace
                        inform user application of successful completion of transaction
                    end
                    Abort:
                    begin
                        inform user application of completion of the abort of T
                    end
                  end-case
                end
            end-case
        until forever
    end. {2PL-TM}
```

The 2PL algorithm can be extended to the distributed DBMS environment. One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager, the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the primary site 2PL algorithm.

The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is shown in figure 6.4, this communication is between the transaction manager at the site where the transaction is initiated (called the coordinating TM), the lock manager at the central site, and the data processor (DP) at the other participating sites. The participating sites are those at which the operation is to be carried out. The order of messages is denoted in the figure.

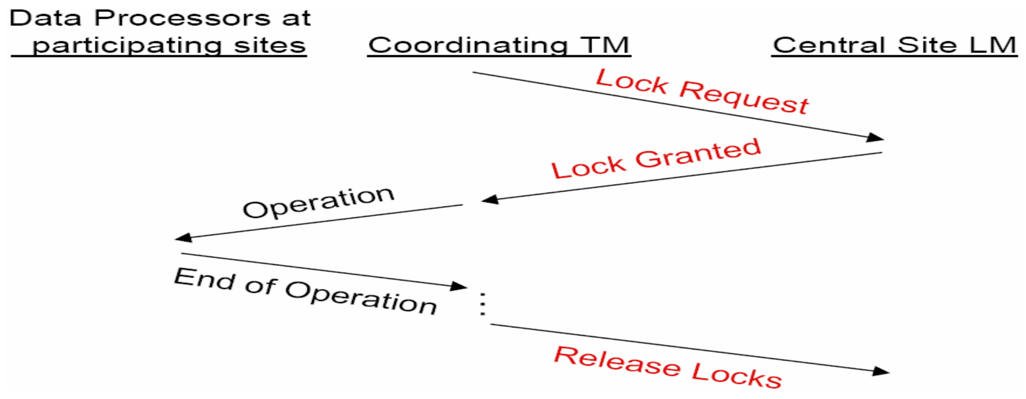


Figure 6.4 Communication Structure of Centralized 2PL

Algorithm 6.3

Algorithm C2PL-TM

declare-var

T : Transaction

Op : Operation

x : DataItem

```
  $msg : Message$

  $O : Dbop$
   $pm : Dpmsg$
   $res : DataVal$
  $S : SiteSet$
begin
    repeat
        WAIT($msg$)
        case of $msg$
          $Dbop :$
          begin
            $Op \leftarrow O.opn$
            $x \leftarrow O.data$
           $T \leftarrow O.tid$
           Case of  Op
              Begin_transaction:
              begin
                  S $\leftarrow \theta$
              end
              Read:
              begin
                  S $\leftarrow$ S $\cup$ {the site that stores x and has the lowest access cost to it}
                  Send O to the central lock manager
              end
              Write:
               begin
                  S $\leftarrow$ S $\cup$ {Si|x is stored at site Si}
                  Send O to the central lock manager
               end
               Abort or Commit:
```

```
  begin
      send O to the central lock manager
  end
end-case
end
Scmsg:                                    {lock request granted on locks released}
 begin
    if lock request granted then
        send O to the data processor in S
    else
        inform user about the termination of transaction
    end-if
  end
  Dpmsg:
  begin
    $Op \leftarrow pm.opn$
    $res \leftarrow pm.result$
    $T \leftarrow pm.tid$
    Case of  Op
        Read:
         begin
             return res to the user application(i.e. the transaction)
         end
        Write:
         begin
              inform user application of completion of the write
         end
        Commit:
         begin
            if commit msg has been received from all participants then
            begin
```

```
                        inform user application of successful completion of transaction
                        send *pm* to the central lock manager
                      **else**                  {wait until commit *msg* comes from all}
                        record the arrival of the commit message
                      **end-if**
                    **end**
                    Abort:
                    **begin**
                      inform user application of completion of the abort of T
                      send *pm* to the central lock manager
                    **end**
                **end-case**
              **end**
          **end-case**
      **until** forever
**end**. {C2PL-TM}
```

Algorithm C2PL-LM

```
**declare-var**
    *msg : Message*
    *dop : SingleOp*
    *Op : Operation*
    *x : DataItem*
    *T : TransactionId*
    *SOP : OpSet*
**begin**
    **repeat**
        WAIT(*msg*)            {the only *msg* that can arrive is from coordinating TM}
          $Op \leftarrow dop.opn$
```

```
 x ← dop.data
T ← dop.tid
Case of  Op
    Read or Write:
     begin
         find the lock unit lu such that x ⊆ lu
         if lu is unlocked or lock mode of lu is compatible, with Op then
     begin
         set lock on lu in appropriate mode
         msg ← "Lock granted for operation dop"
         send msg to the coordinating TM of T
     end
      else
         put Op on a queue for lu
      end-if
   end
   Commit or Abort:
    begin
        for each lock unit lu locked by T do
        begin
            release lock on lu held by T
            if there are operations waiting in queue for lu then
            begin
                SOP ← first operation (call O) from the queue
                SOP ← SOP ∪ {O|O} is an operation on queue that
                                can lock lu in a compatible mode with
                                the current operations in SOP}
                Set the locks on lu on behalf of operation is SOP
                for all the operations O in SOP do
                begin
                    msg ← "Lock granted for operation O"
                    send *msg* to the coordinating TM's
                end-for
            end-if
        end-for
        msg ← "Locks of  T  released"
        send msg to the coordinating TM of T
    end
  end-case
until forever
end. {C2PL-LM}
```

6.3 Deadlock Detection

Any locking based concurrency control algorithm may result in deadlocks, since there is mutual exclusion of access to shared resources (data) and transaction may waits on locks. Therefore, the distributed DBMS requires special procedures to handle them.

Consider two transactions T_i and T_j that hold write locks on two entities x and y [i.e., $wl_i(x)$ *and* $wl_j(y)$]. Suppose that T*i* now issues a $rl_i(y)$ or a wl*i*(y). Since y is currently locked by transaction T*j*, T*i* will have to wait until T*j* releases its write lock on *y*. However, if during this waiting period, T_j now request a lock (read or write) on x, there will be a deadlock. This is because Ti will be blocked waiting for T_j to release its lock on y while for T_j will be waiting for T*i* to release its lock on *x*. In this case, the two transactions T*i* and T_j will wait indefinitely for each other to release their respective locks.

A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place. This outside interference may come form the user, the system operator, or the software system (the operating system or the distributed DBMS).

A useful tool in analyzing deadlocks [2, 4, 14] is a *wait-far graph* (WFG). A WFG is a directed graph that represents the wait-for relationship among transactions. The nodes of this graph represent the concurrent transactions in the system. An arc $T_i \rightarrow T_j$ exists in the WFG if transaction T_j is waiting for T_j to release a lock on some entity. Figure depicts the WFG contains a cycle. We should indicate that the formation of the WFG is more complicated in distributed systems, since two transactions that participate in a deadlock condition may be running at different sites. We call this situation a *global deadlock.* In distributed systems, then, it is not sufficient that each local distributed DBMS form a *local wait-for graph* (LWFG) at each site; it is also necessary to form a *global wait-for graph* (GWFG), which is the union of all the LWFGs.

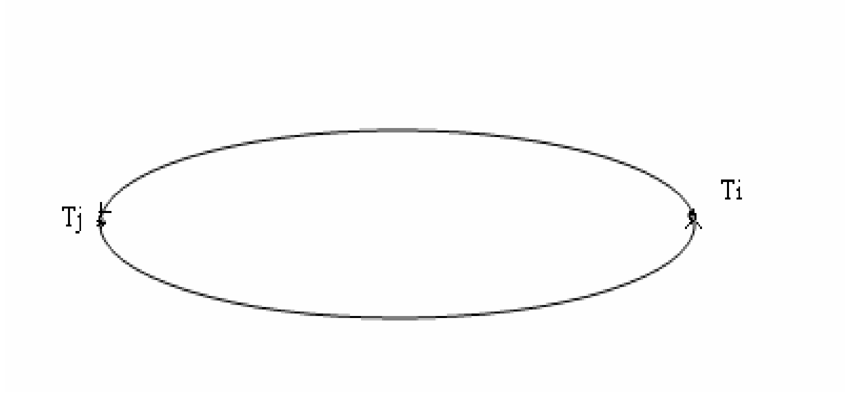


Figure 6.5: A WFG example

Chapter 7

Implementation and Testing

7.1 Implementation

As the implementation part of this study, a simulation environment has been developed. This environment simulates the deadlock detection in distributed database system. The program has been developed using Java programming language (JDK 1.6.0). There has been a distributed system consisting of Lock Manager, Transaction Manager and three Clients.

7.2 Program Structure

The simulation consists of Lock Manager, Transaction Manager and different clients. Lock Manager is centralized and Transaction Manager is distributed over the different sites. The following figure depicts the simple structure of the program.

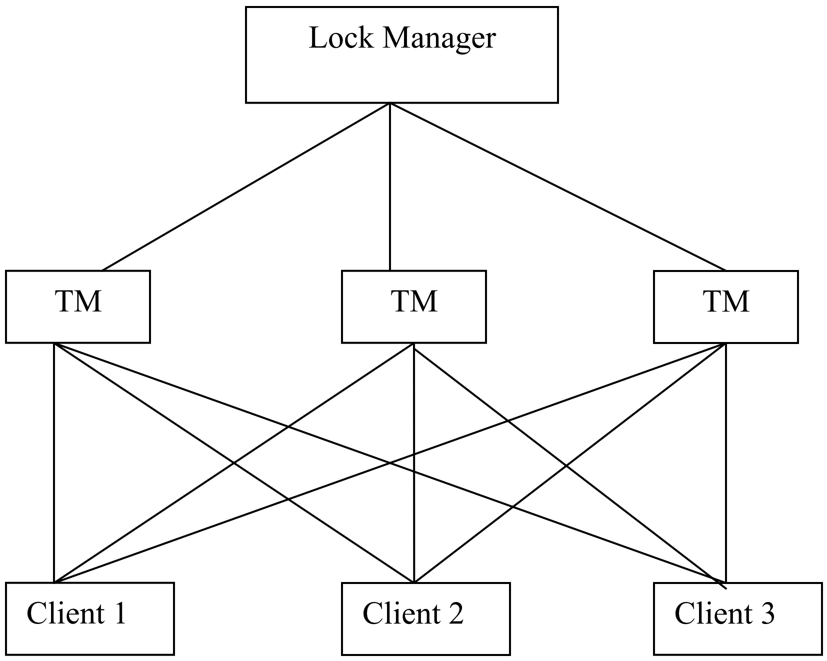


Figure 7.1 Simple Program Structure

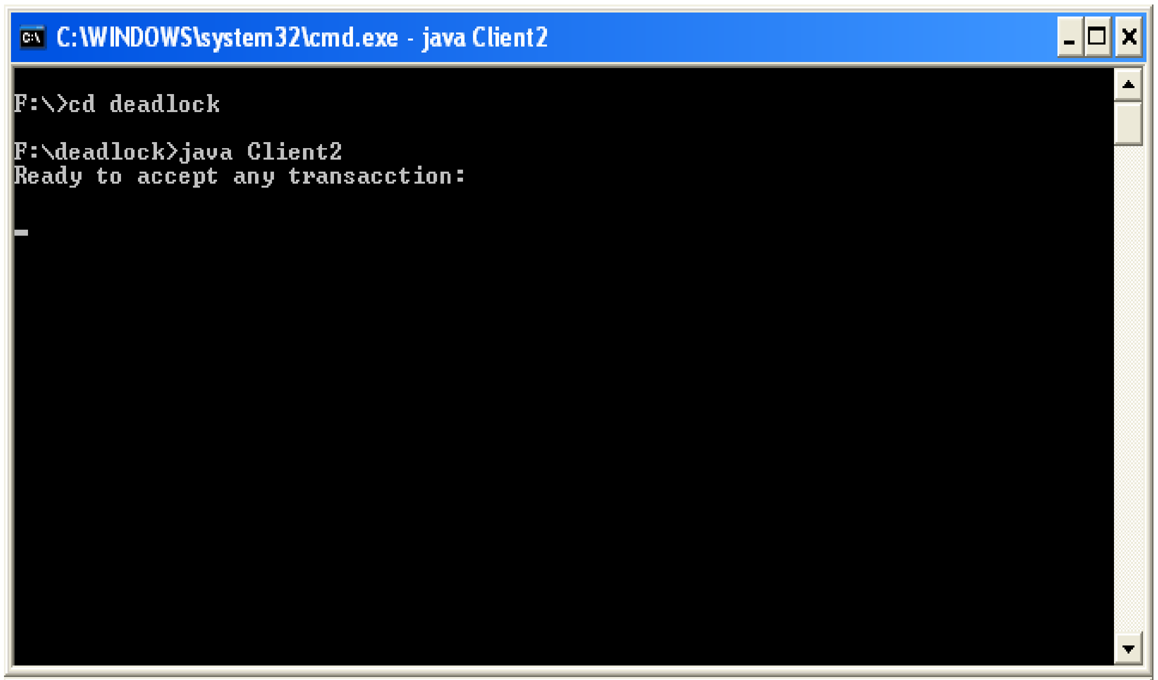
As the main part of the implementation, the centralized two-phase locking Algorithm (presented in previous chapter) has been implemented and by using deadlock detector the deadlock can be detected.

7.3 Testing

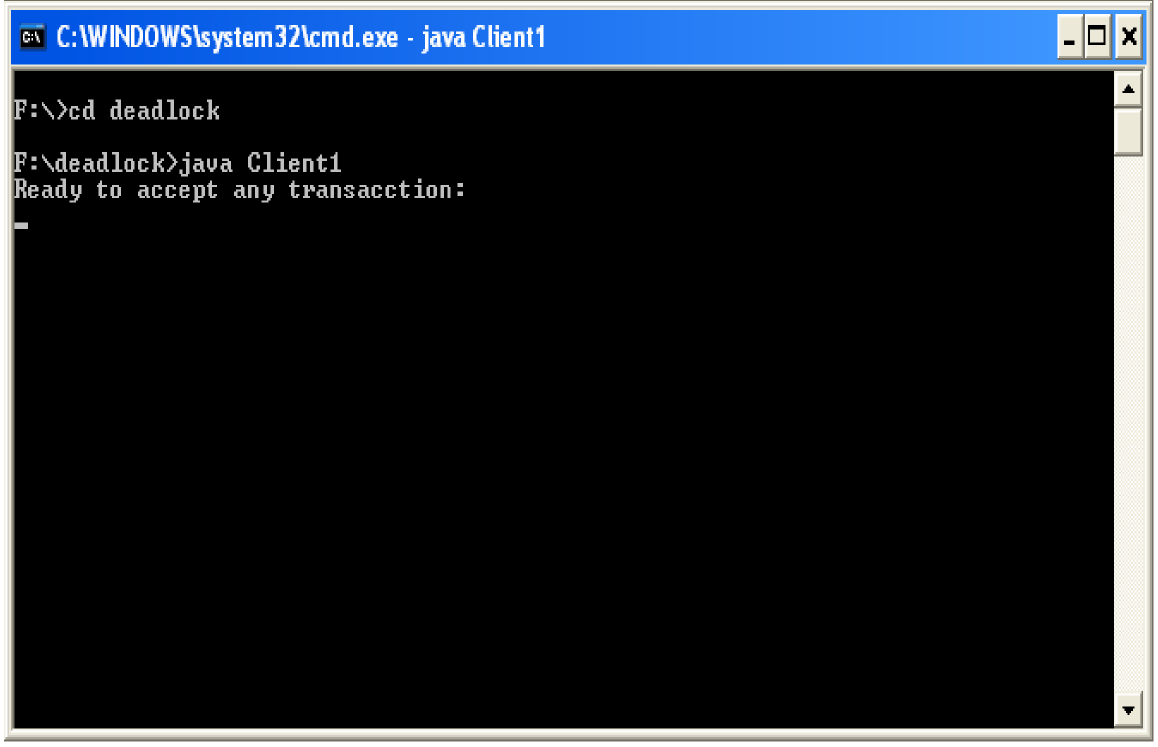
In this section, a sample testing and the result have been described. The testing was carried out on the simulated environment developed in the implementation. The sample program for testing is given below.

Transaction 1	Transaction 2
Begin	Begin
Lock(x)	Lock (y)
Read (y)	Read (x)
………	………..
………	………..
End	End

The two transactions are in deadlock in different clients (client 1 and client 2)

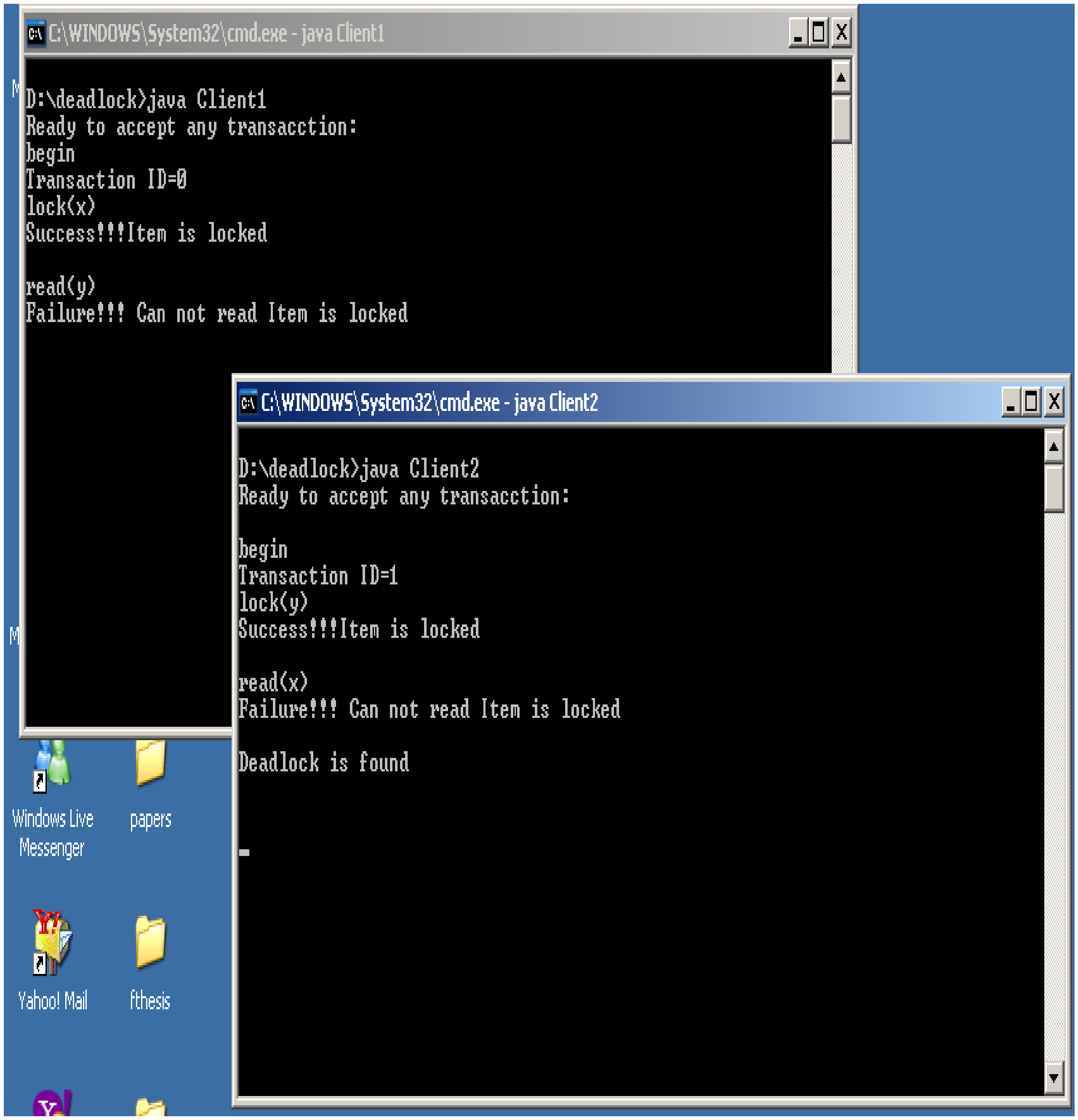


```
F:\>cd deadlock

F:\deadlock>java Client2
Ready to accept any transacction:
```



```
F:\>cd deadlock

F:\deadlock>java Client1
Ready to accept any transacction:
```

Figure 7.2: Client1 and Client2 are ready to accept any transactions.


C:\WINDOWS\System32\cmd.exe - java Client1
Windows Live Messenger
papers
Yahoo! Mail
fthesis
C:\WINDOWS\System32\cmd.exe - java Client2

```
D:\deadlock>java Client1
Ready to accept any transacction:
begin
Transaction ID=0
lock(x)
Success!!!Item is locked

read(y)
Failure!!! Can not read Item is locked
```

```
D:\deadlock>java Client2
Ready to accept any transacction:

begin
Transaction ID=1
lock(y)
Success!!!Item is locked

read(x)
Failure!!! Can not read Item is locked

Deadlock is found
```

Figure7.3 : Client1 and Client2 are in deadlock

7.4 Output Analysis

The output of the program execution has been shown in the above java screen (figure 7.2 and 7.3). Here, in client 1 a transaction lock the data item x and ready to read the data item y and transaction complete. In client 2 a transaction lock the data item y and ready to read the data item x. Here deadlock occurs because transaction 1 is ready to read the data item y but that is already locked by transaction 2 and transaction 2 is ready to read the data item x that is already locked by transaction 1. So, deadlock is found between transactions of different clients on data item x and y.

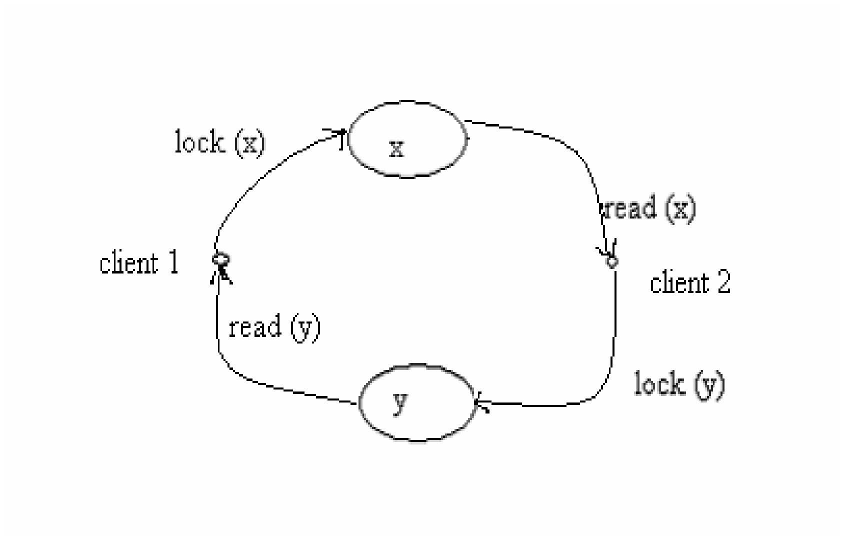


Figure 7.4 deadlock in two clients

Chapter 8

Conclusion and Further Recommendations

8.1 Conclusion

The study was mainly focused the two-phase locking and deadlock detection in a distributed database system. This study describes the transaction processing and the concurrency control problem in distributed system. The concurrency control problem is exacerbated in a distributed database management system because users may access data stored in many different computers in a distributed system. This study includes different types of problems such as: dirty read problem, lost update problem, fuzzy read problem, phantom problem etc. These problems arise due to interference among conflict operations of concurrent transactions. So, the seralizability is required for the solution of interference problems in concurrent environment. Seializability controls the execution order of conflicting operations and ensures correction of concurrent execution.

To preserve the consistency of distributed database, this study adopts locking mechanism. This study focused on C2PL algorithm, where LM is centralized and TM is distributed over the different clients. In this study, LM checks the different clients for particular data item to detect a deadlock or not. Deadlock occurs due to if any transaction read the data item that has already locked by another transaction.

8.2 Further Recommendations

This study has certain limitations which can be fulfill by further study. This study specially focused on lock based concurrency control algorithm and deadlock detection in distributed database system. This study can be extended to solve deadlock problem by using Timestamp Ordering (TO). This study could not be evaluated in heterogeneous database system. This study can be extended to examine deadlock detection and prevention in heterogeneous database system.

REFERENCES

[1] Agarwal, R. and DeWitt, D. (1985), "*Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation*", ACM TODS, Vol. 10, pp. 529-564.

[2] Agarwal, R., Carey, M. J. and McWoy, L. (1988?), *"The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems"*, IEEE Trans. Software Engg.

[3] Agarwal, R., Carey, M. J. and Livny, M. (1987), "*Concurrency Control Performance Modeling: Alternatives and Implications*", ACM TODS, Vol. 12, pp. 609-654.

[4] Alexander Thomason "*Concurrency Control: Methods, Performance, and Analysis*" IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532

[5] Bayer, R., II. Iieller, and A. Reiser, *"PamllelIsm and Re&very In Database Systems,"* ACM trans. on Database Systems.

[6] Bernstein, P. A. and Goodman, N. (1980), "*Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems*", Proc VLDB, Oct. 1980, pp. 285-300.

[7] Bernstein, P. A. and Goodman N. (1981), "*Concurrency Control in Distributed Database Systems*", ACM Computing Surveys, June 1981, pp. 185-222.

[8] Bernstein, P. A., and Goodman, "*N. Fundamental algorithms for concurrency control in distributed database systems*". Tech. Rep., Computer Corp. America, Cambridge, Mass., 1980.

[9] Barghouti N.S., Kaiser G.E. "*Concurrency Control in Advanced Database Applications"*, ACM Computing Survey 23, 3 (Sept), 269-317.1991

[10] Badal, D.Z. "*Correctness of Concurrency Control and Implications in Distributed Database*" Proc.COMPSAC 79 Conf., Chicago, Nov. 1979.

[11] Bharat Bhargava "*Concurrency Control in Database Systems*", Transactions on knowledge and Data Engineering, vol. 11, no. 1, January/February 1999

[12] Carey, M. and Stonebraker, M. (1984), "*The Performance of Concurrency Control Algorithms for Database Management Systems*", In VLDB 1984, pp. 107-118.

[13] Chandy, K. M., Misra, J., and Haas, L. "*Distributed deadlock detection*". ACM Trans. Comput." Syst. I,2 (May 1983), 144-156.

[14] Coffman, E., Elphich, M., Shoshani, A. (1971), "*System Deadlocks*", ACM Computing Surveys, 2, 3, pp. 67-78.

[15] C. Mohan, B. Lindsay, and R. Obermarck "*Transaction Management in the R* Distributed Database Management System*" IBM Almaden Research Center.

[16] Carolyn Mitchell, "*Components of a Distributed Database*", Norfolk State University, Department of Computer Science, Technical Report #NSUCS-2004-005

[17] C. Papadimitrion "*The Theory of Database Concurrency Control*", Computer Science Press. 1986

[18] Donald W. Larson," *Central vs. Distributed Systems*",dwlarson@mac.com, http://www.timeoutofmind.com

[19] Elmagarmid A.K. et al., "*Database Transaction Models for Advanced Applications"* (Morgan Kaufmann, 1992).

[20] Elmasari R., Navathe S B, "*Fundamentals of Database Systems",* Third Edition, Pearson Education.

[21] Gray, J. (1981), "*The Transaction Concept: Virtues and Limitations*", Proc VLDB, Sept 1981.

[22] Gray, J. N., Lorie, R. A. and Putzolu, G. R.(1975), "*Granularity of Locks in a Large Shared Data Base"*, Proc VLDB, Sept 1975.

[23] Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L. (1976), "*Granularity of Locks and Degrees of Consistency in a Shared Data Base*", in Proc. IFIP TC-2 Working Conference on Modelling in Data Base Management Systems, Ed. G. M. Nijssen, North-Holland.

[24] Gligor, V. D., and Shattuck, S. H. "*Deadlock detection in distributed systems*". IEEE Trans. Softw. Eng. SE-6, 5 (Sep. 1980), 435-440.

[25] Gray J. and Reuter A., "*Transaction Processing: Concepts and Techniques",* CA

[26] Gray, J. N. "*The transaction concept: Virtues and limitations*", In Proc. 7th Znt. Conf. Very Large Data Bases (Sept. 1981), 144-154.

[27] Garcia-Mona, H. "*Using Semantic Knowledge For Transaction Processing In A Distributed Database*", ACM Transaction on Database Systems*, Vol 8, No. 2,* June, 1983.

[28] H. T. Kung, J.T.Robinson *"On Optimistic Methods for Concurrency Control*", *ACM Transactions on Database Systems, VO1.6,* no.2, 1981, pp.213-226.

[29] J. Gray, R. Lorie, G. Putzulo, and I. Traiger, "*Granularity of Locks and Degrees of Consistency in a Shared Database*", IBM Research Report RJ1654, September 1975.

[30] Kjell Bratbergsengen **"***Concurrency Control in Distributed Object-Oriented Database Systems*" Department of Computer and Information Science Norwegian University of Science and Technology Trondheim, Norway

[31] L.Sha, R. Rajkumar, J .Lehoczky: *"Concurrency Control for Distributed Real-Time Databases"*, *ACM SIGMOD Record,* March 1988

[32] Lin, W. T. K., and Nolte, J. "*Performance of two phase locking*". In Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks (Feb. 1982), 131-160

[33] Lin, W. T. K., and Nolte, *" Basic timestamp, multiple version timestamp, and two phase Locking*", Computer Corp. America, Cambridge, Mass., Jan. 1983

[34] Menasce, D., and Muntz, R. "*Locking and deadlock detection in distributed data bases*". IEEE Trans. Softw. Eng. SE-& 3 (May 1979), 195-202.

[35] M. Casanova "*The Concurrency Control Problem for Database Systems*", Ph.D. Thesis, Computer Science Department, Harvard University.1979.

[36] Ozsu M T and Valduriez P, "*Principles of Distributed Database Systems",* (Prentice-Hall, 1991).

[37] Obermarck, R. "*Distributed deadlock detection algorithm*". ACM Trans. Database Syst. 7, 2 (Jun. 1982), 187-208.

[38] Papadimitriou, C. H. (1986), "*The Theory of Database Concurrency Control*", Computer Science Press.

[39] P. A. Bernstein, V. Hadzilacm, N.Goodman: *"Concurrency Control and Recovery in Database Systems,"* Addison-Wesley, 1987.

[40] Partha Dasgupta d Zvi M Kedem," *A Non-Two-Phase Locking Protocol for Concurrency Control in General Databases*". Department of Computer Science State University of New York Stony Brook, NY 11794

[41] Prof. Sin-Min Lee "*Concurrent Control Using 2-Phase Locking*" Department of Computer Science

[42] Phlip A. Bernstein "*Transaction Processing*", 1999.

[43] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M., II. 1978. "*System level concurrency control for distributed database systems." ACM Trans. Database Syst. 3,* 2 (June)

[44] Ries, D. R. and Stonebraker, M. R. (1977), "*Effects of Locking Granularity in a Database Management System*", ACM TODS, Vol 2

[45] Ries, D. R. "*The effects of concurrency control on database management system performance*", Ph.D. dissertation, Computer Sciences Dep., University of California, Berkeley, April 1979.

[46] Svetlana Vasileva, Petar Milev, Borislav Stoyanov: *"Some Models of a Distributed Database Management System with Data Replication"*

[47] Silberschatz A, Korth F H, Sudarshan S, "*Database System Concepts"*, Fourth Edition, McGraw-Hill Higher Education, 2002.

[48] Traiger, I. L., Gray, J. N., Galtieri, C., and Lindsay, B. G. "*Transactions and consistency in distributed database systems*", Rep. RJ2555, IBM Research Lab., San Jose, Calif., June 1979.

[49] Wei Chen "*A prototype of distributed database system with 2-phase locking algorithm*" (wechen11@vt.edu) April 29, 1999