CHAPTER 2

OVERVIEW OF TRANSACTION MANAGEMENT

This chapter introduces the principles of database systems with centralized and distributed architectures. Next, the basic concept of a transaction which is a foundation for concurrent execution and recovery from system failure in a DBMS is discussed. Finally, the concurrent control mechanisms and distributed transaction management are discussed.

2.1 Database Architecture

A database is a repository of facts about the organization it models [2]. The state of the organization continuously changes and these changes must be incorporated into the database to preserve the facts. The system uses a transaction mechanism for incorporating these changes into the database in a consistency preserving manner. A database management system (DBMS) is a collection of interrelated data and a set of programs which manages transactions to access those data. A query language such as SQL is used by users to interact with the DBMS. The transactions submitted by the users are executed under the concurrency control mechanism and after necessary updation the results are sent to the user by the DBMS. Two types of architectures are available for implementing DBMS: (a) centralized (b) distributed.

2.1.1 Architecture of Centralized DBMS

It consists of a server and many clients. The clients are connected to the server through wired network. User requests are submitted to a client using a query language
such as SQL. If it has the necessary resources, this query is processed by the client. Otherwise it is sent to the server for processing. The results are sent to the client by the server. The client may format the results and send them to the user. The architecture of centralized DBMS is shown in Figure 2.1.

2.1.2 Architecture of Distributed DBMS

It consists of a number of centralized database systems which communicate with one another through various communication media such as high speed network or telephone lines. The database systems that run on each site may have mutual independence. Each site may participate in the execution of transactions that access data at one site or several sites. In contrast to centralized database systems where the data reside in one single location, in a distributed database systems the data reside in several locations. Many difficulties arise in transaction processing due to the distribution of data. Distributed databases are classified as homogeneous and heterogeneous as discussed below.

- **Homogeneous Distributed Database System:** In this environment, all sites will have the same DBMS software.

- **Heterogeneous Distributed Database System:** In this environment, different sites run under the control of different DBMSs autonomously and are connected to allow users to have data access from multiple sites. The key to building heterogeneous system is to have well-accepted standards for gateway protocols that expose DBMS functionality to external applications. The differences between servers are bridged by the use of gateway protocols. The heterogeneous system is also known as multi database system. Figure 2.2 illustrates the architecture of distributed DBMS.
Figure 2.1: Architecture of centralized DBMS

Figure 2.2: Architecture of distributed DBMS
Distributed Data Storage

Relations are stored at several sites in a distributed database. In order to minimize the message passing costs needed to access a relation stored at a remote site, a single relation may be partitioned or fragmented across several sites. In fragmentation, the system partitions the relation into several fragments and stores each fragment at a different site. In horizontal fragmentation, each fragment consists of a subset of rows of the original relation. In vertical fragmentation, each fragment consists of a subset of columns of the original relation. If a relation is replicated, a copy of the relation is stored in two or more sites. A relation is fully replicated if its copy is stored in every site.

Database Distribution in Distributed DBMS

Distribution of data in a distributed system can be done in three ways as discussed below.

Database Partition: The entire database is divided into many partitions. Using the following criteria, database partitions are allocated to database servers.

(i) The majority of the queries can be processed by the partition at that server (support for highest database locality).

(ii) Partitioning should minimize the data communication cost, global consistency maintenance cost and the recovery cost.

(iii) It should help to localize the serialization of concurrent transactions.

The possibility of a centre point of failure makes the partition scheme less reliable. When a server fails, its partition becomes unavailable until it recovers from the failure.
Partial Replication: Apart from database partitioning, this scheme allows replication of a subset of partitions at more than one server. The partial replication has all the properties of a partition scheme and it further improves database locality and reliability. Even if one server fails, the replicated copy of the partition available in another server helps in processing queries. The recovery is also easier in the partial replication. It is relatively more time consuming to maintain the global consistency because any change in a partition must be installed to all its replications located at various servers. To further increase database locality the database is fully replicated.

Full Replication: Under this scheme the entire database is replicated at all servers. This has maximum locality and it also minimizes data communication cost during query processing. The fully replicated scheme provides the highest reliability and availability but it happens at the cost of maintaining global consistency.

2.2 Transaction Concept

A transaction is a unit of program execution that accesses and possibly updates various data items [7]. A transaction facilitates consistency preserving changes. A transaction, therefore, transforms the database from an initial consistent state to the next consistent state using a set of operations which are applied on the data items. A transaction consists of operations which are enclosed by begin transaction and end transaction statements.

2.2.1 States of a Transaction

A transaction will go through the following states during its execution [7].

- **Active:** A transaction is in active state during its execution.
25

- **Partially committed**: A transaction comes to partially committed state once the final statement is executed.

- **Failed**: A failed state is reached after the discovery that normal execution can no longer proceed.

- **Aborted**: The execution of the transaction has failed and this execution does not have any effect on the initial consistent state of the database.

- **Committed**: The transaction is said to be committed if it has successfully completed its execution.

### 2.2.2 ACID Properties

A DBMS must maintain the following four properties of transactions in order to ensure integrity of data [8].

- **Atomicity**: Either all operations of the transaction are carried out or none. Incomplete transactions should not have any effect on the state of a database in case of failures.

- **Consistency**: Each transaction execution in isolation with no concurrent execution of other transactions should preserve the consistency of the database.

- **Isolation**: This property makes sure that to preserve consistency, two conflicting operations (for example, read and write or write and write) on the same data item by two different transactions are not allowed. The isolation property is enforced by means of a concurrency control mechanism or serialization mechanism.

- **Durability**: After successful completion of a transaction its effect should persist even if the system fails before all the changes are reflected on disk.
Transaction consistency has to be ensured by users. The next state of the database is consistent only if the previous state was consistent. A transaction takes it for granted that it always works on consistent database and it also guarantees that it will produce a consistent database state. The isolation property is achieved by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. The net effect of concurrent execution of two transactions T1 and T2 is guaranteed to be equivalent to executing T1 followed by executing T2 or executing T2 followed by executing T1.

Transaction atomicity is ensured by undoing the actions of incomplete transactions. For this purpose, a log consisting of all writes to the database is maintained by the DBMS. The log is also useful to ensure durability. A system failure can occur before the updates made by the transaction are written to disk permanently. In that case, the log is used to remember and restore those changes when the system restarts.

2.2.3 Serialization of Transactions

In centralized systems, the transaction processing is carried out entirely at one location. The transactions initiated by the clients are submitted to the server for execution. The server sends the results to the clients after completing the concurrent transaction execution. The cost of meeting ACID constraints and database recovery is very low due to the concurrent processing of transactions. But this results in low system throughput. For the cases where transactions are mostly short-lived and the workload is low, this environment is acceptable.
Consistency preserving execution is achieved by means of serial execution. In serial execution, the next transaction begins only when the last transaction terminates (commits or aborts) to ensure that data sharing does not take place. Though serial execution is advantageous, it has a side effect of poor resource utilization which results in low efficiency (poor response time and throughput) of the database system, thereby increasing the cost of achieving the consistency significantly.

The solution to this problem is concurrent execution of transactions with controlled data sharing. This is done using a process called serialization of concurrent transactions, which mutually excludes or isolates two transactions over the use of common data item.

### 2.2.4 Serializability

In this section, we discuss two types of schedule equivalence which leads to the concepts of conflict serializability and view serializability [9, 10]. If the actions of different transactions are not interleaved – that is, transactions are executed from start to finish, one by one – we call the schedule a serial schedule. An example for a serial schedule is given in Figure 2.3. A serializable schedule over a set $S$ of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over $S$. An example for a serializable schedule is given in Figure 2.4. In this schedule, even though the actions of $T_1$ and $T_2$ are interleaved, the result is equivalent to running $T_1$ and then running $T_2$. $T_1$’s read and write of $B$ is not influenced by $T_2$’s actions on $A$. The net effect is the same if these actions are swapped to obtain a serial schedule $T_1; T_2$ as given in Figure 2.3.
Different orders of scheduling transactions serially may produce different results but they are acceptable. Another serializable schedule (obtained from the two example transactions from Figure 2.3) can be interleaved as shown in Figure 2.5. This schedule is equivalent to the serial schedule T2;T1 as shown in Figure 2.6.
Anomalies due to Interleaved Execution

There are three main ways in which a schedule involving two consistency preserving committed transactions could run against a consistent database and leave it in an inconsistent state. Two actions by two transactions on the same data item are said to conflict if at least one of them is write operation. There are three cases when the actions of two transactions T1 and T2 conflict with each other. In a write-read
(WR) conflict, T2 reads a data item previously written by T1. In read-write (RW) conflict, T2 writes a data item previously read by T1. Similarly, we can define a write-write (WW) conflict.

- **WR conflict:** Transaction T2 reads a data item A that has been modified by another transaction T1 which has not yet been committed. This read is called dirty read.

- **RW conflict:** Transaction T2 changes the value of a data item A that has been read by a transaction T1 while T1 is still executing. If T1 tries to read the value of A again, a different result will be got even though it has not modified A in the meantime. This read is known as unrepeatable read.

- **WW conflict:** Transaction T2 overwrites the value of the data item A which has already been modified by a transaction T1 while T1 is still executing. Even if T2 does not read the value of A written by T1 there can be a potential problem. This is known as lost update.

**Conflict Serializability**

Two schedules are conflict equivalent if they involve the same set of actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way. Two actions are said to conflict, if they operate on the same data item and at least one of them is write. The order of conflicting operations determines the outcome of a schedule. Any pair of non-conflicting operations can be interchanged without altering the effect of the schedule on the database. By repeatedly swapping pairs of non-conflicting actions, one of the schedules can be got since they order all pairs of conflicting operations in the same way. Thus if a schedule S can be transformed into a schedule S’ by a series of swaps
of non-conflicting actions, we say that S and S’ are conflict equivalent. Schedule 1 of Figure 2.3 is conflict equivalent to schedule 2 of Figure 2.4 since the read(B) and write(B) instructions of transaction T1 can be swapped with the read(A) and write(A) instruction of T2.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule is conflict serializable if it is conflict equivalent to a serial schedule. Thus, schedule 2 is conflict serializable since it is conflict equivalent to the serial schedule 1. Schedule 5 of Figure 2.7 is not conflict serializable since it is not equivalent to either the serial schedule T3; T4 or the serial schedule T4;T3.

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td>write(Q)</td>
</tr>
<tr>
<td>write(Q)</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2.7: Schedule 5*

**View Serializability**

View Serializability is less stringent than conflict equivalence. Consider two schedules S1 and S2, where the same set of transactions participates in both schedules. The schedules S and S’ are view equivalent if three conditions are met [7].

1. For each data item Q, if transaction Tᵢ reads the initial value of Q in schedule S1, then transaction Tᵢ must, in schedule S2, also read the initial value of Q.
2. For each data item Q, if transaction Tᵢ executes read(Q) in schedule S1, and if that value was produced by a write(Q) operation executed by transaction Tᵢ,
the read(Q) operation of transaction Tᵢ must, in schedule S2, also read the value of Q that was produced by the same write(Q) operation of transaction Tᵢ.  

3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S1 must perform the final write(Q) operation in schedule S2.

Conditions 1 and 2 ensure that each transaction read the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2 ensure that both schedules result in the same final system state.

<table>
<thead>
<tr>
<th></th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td></td>
<td>write(Q)</td>
<td></td>
</tr>
<tr>
<td>write(Q)</td>
<td></td>
<td></td>
<td>write(Q)</td>
</tr>
</tbody>
</table>

*Figure 2.8: Schedule 6 - a view serializable schedule*

The concept of view equivalence leads to the concept of view serializability. A schedule is view serializable if it is view equivalent to some serial schedule. Schedule 6 shown in Figure 2.8 is view serializable. It is view equivalent to the serial schedule T₃;T₄;T₅, since the one read(Q) instruction reads the initial value of Q in both schedules and T₅ performs the final write of Q in both schedules.

### 2.2.5 Recoverability

Database recovery protocols recover a database from transaction or system failures, that is, they restore the database to a consistent state from where transaction
processing resumes. These failures may occur due to a number of reasons such as addressing error, wrong input, RAM failure, etc. In a concurrent execution environment when a failure occurs then a transaction may be active or blocked or being rolled back or in the middle of a commit. The task of a recovery protocol is to identify the right operation for recovery for each transaction. These operations are (a) Roll forward or Redo and (b) Roll backward or Undo. Depending upon the execution status of a transaction, one of these operations is selected. Thus, in a recovery process some transactions are undone and some transactions are redone. To implement these operations, Transaction log is required, which is generated and maintained by the system. The log contains committed values of data items (Before Image – BFIM) and modified values of data items (After Image – AFIM). The log is a crucial document for recovery; therefore, it is generated and maintained by a protocol called Write Ahead Logging – WAL. The protocol guarantees that the contents of a log is reliable and can be used for Undo and Redo operations.

After a failure the database system reboots and, by using log, applies Redo and Undo operations on transactions which were in the system when it failed. A redo completes the commit operation for a transaction, and an Undo rolls back a transaction to maintain atomicity.

2.3 Concurrency Control Mechanisms

Concurrency control mechanisms are used for guaranteeing serialization and as a consequence to ensure transaction isolation. Serialization is done by enforcing an execution order for conflicting operations from two different transactions in such a way that it is equivalent to a serial execution of these transactions. Concurrency control mechanism is responsible for maintaining serializability either by rolling back
or by blocking one of the conflicting transactions. The Concurrency Control mechanisms are categorized into approaches: (a) Locking (b) Non-locking. Locking approaches use two phase locking protocol [11] for the purpose of locking data items for transactions and rollback and blocking operations for resolving conflicts over shared data items [11, 12]. These can be applied by using different combinations which can result in a number of concurrency control mechanisms.

2.3.1 Locking Schemes

Two phase locking protocol allows a transaction to issue lock and unlock requests in two phases. In the first phase (Growing phase), a transaction locks the required data objects before their use. In the second phase ( Shrinking phase), a transaction releases its locked data objects but may not obtain any new locks. Four different combinations are used for managing two phase locking. Each combination uses locking, execution and unlocking [12] as three phases as explained below.

- **Simultaneous lock and Simultaneous release:** This scheme allows locking, execution and unlocking to take place atomically. After the successful completion of the previous phase, the next phase starts. The execution life of a transaction is defined as start and end of locking = start and end of execution = start and end of unlocking. Failure that occurs in one phase will not affect other phases. For example, if there is a failure in the execution phase, it can be restarted without restarting the locking phase.

- **Incremental locking and Simultaneous release:** This scheme allows interleaving of locking and execution phases which precede the entire unlocking phase. Transaction execution can be defined as lock → process → lock → process = unlock. Thus, in the growing phase, the transaction locks
and accesses only those data items it actually processes. Locks are released simultaneously once the execution phase is over. Deadlock can occur in this scheme as a side effect. But this side effect can be managed using efficient ways.

- **Simultaneous locking and Incremental release:** In this scheme, growing phase is atomic and unlocking phase is interleaved with execution phase. The execution of a transaction can be defined as locking → execution → unlock → execution → unlock. The drawback of incremental release is cascading which consumes a lot of time.

- **Incremental locking and Incremental release:** Interleaving of execution phase with locking and unlocking phases is done in this scheme. Thus deadlock and cascading can occur in this scheme. Even though it is possible to reduce transaction waiting time, the high cost of handling cascading makes this scheme practically not usable.

When there is a conflict between two transactions over a data item, the concurrency control mechanism has to decide which transaction should be blocked or rolled back to allow the other transaction to continue. The conflict resolution methods have a significant effect on the performance of concurrency control mechanisms. The main objective of an efficient concurrency control mechanism is to make the best possible selection in such a way that the effects of rollbacks and blocking are minimal on the system throughput. This selection procedure is done either by selecting a victim on an ad-hoc basis or by applying some guess work. Conflict resolution methods can be improved by using some intelligence while resolving conflicts as illustrated in [12, 13, 14, 15, 16, 17, 18].
2.3.2 Non-Locking Schemes

Serialization of concurrent transactions through locking is the most commonly used concurrency control scheme. It, however, generates locking overhead. A few thousand instructions are required to be executed each time in order to lock and unlock a data item. Some of the incremental locking schemes generate additional overhead due to deadlock detection and resolution. These overheads are eliminated in timestamp-based schemes.

Time Stamps

The timestamp approach for serializing the execution of concurrent transactions was developed for the purpose of more flexibility to eliminate the cost of locking and to cater for distributed database systems [19, 12]. In timestamping, the execution order of concurrent transactions is defined before they begin their execution. The execution order is established by associating a unique timestamp TS(T_i) to every transaction T_i. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j). There are two simple methods to implement this scheme. The first method uses the value of the system clock as the timestamp. The second method uses a logical counter which is incremented after a new time stamp has been assigned.

Serializability order is determined by the timestamps of the transactions. Therefore, if TS(T_i) < TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.
The Time Stamp Ordering Protocol

To implement this concurrency control scheme, every data item O is given a read timestamp RTS(O) and write timestamp WTS(O). If transaction T wants to read data item O, and TS(T) < WTS(O), the order of this read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted with a new and larger timestamp. If TS(T) > WTS(O), T reads O, and RTS(O) is set to the larger of RTS(O) and TS(T).

If transaction T wants to write data item O, this protocol operates as follows.

1. If TS(T) < RTS(O), the write action conflicts with the most recent read action of O, and T is, therefore, aborted and restarted.
2. If TS(T) < WTS(O), a naive approach would be to abort T because its write action conflicts with the most recent write of O and is out of timestamp order. However, we can safely ignore such writes and continue.
3. Otherwise, T writes O and WTS(O) is set to TS(T).

When two transactions conflict over a data item, their timestamps are used to enforce serialization by rolling back one of the conflicting transactions. Even though, the time stamping technique is useful to eliminate locking and deadlock management costs, it restricts the conflict resolution options compared to two phase locking schemes. Moreover, it also adds some new overheads which can affect system performance more than locking approaches [16].

Mixed approaches

To exploit the dynamic aspects of two phase locking and the static ordering of timestamping, a number of concurrency control techniques were developed by using a
combined approach. In mixed approach techniques called Wound-wait and Wait-die [12, 21], the enforcement of mutual exclusion among transactions is carried out using locking while conflicts are resolved using timestamps.

- **Wound-Wait (WW):** In Wound-Wait (WW) a conflict is resolved by rolling back a younger (larger timestamp) holder. This is a preemptive algorithm since it rolls-back the holder which might be under execution or waiting for a data item locked by an older (smaller timestamp) transaction. It avoids deadlock by not blocking the older requester. When there is a conflict, then

\[
\text{If } T_j \text{'s timestamp } > T_i \text{'s timestamp, then} \\
T_j \text{ waits for } T_i \text{ to terminate (commit or abort)} \\
\text{else } T_i \text{ is wounded } (T_j \text{ forces } T_i \text{ to abort})
\]

- **Wait-Die (WD):** In Wait-Die (WD) action is taken only on the requester. A requester is rolled back if it is younger than the holder, otherwise it is blocked. It is a non-pre-emptive algorithm since when a conflict over an entity occurs, it never takes away any resource from the holder (younger or older) in conflict. It avoids deadlock since it rolls back a younger requester only which is capable of precipitating a deadlock.

\[
\text{If } T_j \text{'s timestamp } > T_i \text{'s timestamp, then} \\
T_j \text{ dies (rolled back)} \\
\text{Else } T_j \text{ waits for } T_i \text{ to terminate (commit or abort)}
\]

**Optimistic Concurrency Control Schemes**

In [67], Optimistic concurrency control scheme is presented. This scheme reduces locking overhead by delaying lock operation until conflicting transactions are ready to commit. They rely on efficiency in the hope that conflicts between
transactions will not occur. Since transaction execution is deadlock free, this scheme is called optimistic concurrency control or certifiers [19, 23]. Transactions execute in three phases as explained below.

- **Read:** The transaction reads values from the database and stores in a local cache.

- **Validate:** At commit time, the Data Base Management System checks for any conflict with any other concurrently executing transaction. In case there is a conflict, the transaction is aborted and the local cache is cleared and it is restarted.

- **Write:** In this phase, if the validation is successful, the transaction copies the updated data items into the database.

This scheme’s performance is better than the locking scheme if there are few conflicts and the validation can be done efficiently. In the event of many conflicts, the performance suffers significantly due to the cost involved in repeated restarting of transactions.

At the start of the validation phase, each transaction is assigned a timestamp. Then validation phase checks to find out whether the timestamp ordering of transactions is an equivalent serial order. For every pair of transactions $T_i$ and $T_j$ such that $TS(T_i) < TS(T_j)$, one of the validations conditions must hold.

1. $T_i$ completes all the three phases before $T_j$ begins.
2. $T_i$ completes before $T_j$ starts its write phase, and $T_i$ does not write any data item read by $T_j$.
3. $T_i$ completes its Read phase before $T_j$ completes its Read phase, and $T_i$ does not write any data item that is either read or written by $T_j$. 


To validate $T_j$, we must check to see that one of these conditions holds with respect to each committed transaction $T_i$ such that $\text{TS}(T_i) < \text{TS}(T_j)$. Each of these conditions ensures that $T_j$’s modifications are not visible to $T_i$.

The first condition allows $T_j$ to see some of $T_i$’s changes, but the transactions execute completely in serial order with respect to each other. The second condition allows $T_j$ to read data items while $T_i$ is still modifying data items, but there is no conflict because $T_j$ does not read any data item modified by $T_i$. Although $T_j$ might overwrite some data items written by $T_i$, all of $T_i$’s writes precede all of $T_j$’s writes. The third condition allows $T_i$ and $T_j$ to write data items at the same time and thus have even more overlap in time than the second condition, but the sets of data items written by the two transactions cannot overlap. Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met.

**Multiversion Concurrency Control**

In multiversion approach discussed in [19], the main goal is to reduce the requester’s waiting time needed to read a data item. Thus this approach immediately fulfils the data requests of a transaction. Timestamp-based multiversion concurrency control scheme [24] maintains many versions of each data item, each with a write timestamp, and let transaction $T_i$ read the most recent version whose timestamp precedes $\text{TS}(T_i)$.

If a transaction $T_i$ wants to write a data item, we must ensure that the data item has not already been read by some other transaction $T_j$, such that $\text{TS}(T_i) < \text{TS}(T_j)$. If $T_i$ is allowed to write such a data item, its change should be seen by $T_j$ for serializability, but $T_j$, which read the data item at some time in the past, will not see $T_i$’s change.
For the purpose of checking this condition, every data item has an associated read time stamp. Whenever a data item is read by a transaction, the read timestamp is set to the maximum of the current read timestamp and the reader’s timestamp. If \( T_i \) wants to write a data item \( O \) and \( TS(T_i) < RTS(O) \), \( T_i \) is aborted and restarted with a new larger timestamp. Otherwise, a new version of \( O \) is created by \( T_i \). \( T_i \) also sets the read and write timestamps of the new version to \( TS(T_i) \).

The Timestamp based multiversion concurrency control scheme has the desirable property that a read request never fails and is never made to wait. This scheme is more applicable in typical database systems, where read operations are more frequently used than write operations.

### 2.4 Distributed Transactions

In a distributed system, a transaction is divided into a number of subtransactions. These subtransactions are distributed to suitable processing nodes (servers) for concurrent processing or parallel processing. The resource waste is minimized during distributed transaction processing. But special approaches are required for managing the work load (concurrency, recovery, etc.).

In a distributed environment, there are two types of transactions: (a) local transactions that execute and access data in only one local database (b) global transactions that execute and access data in several local databases. ACID properties of local transactions are maintained by using various schemes as described in the previous section. Since many sites may be participating in the execution of global transactions, the task of ensuring ACID properties is much more complex.
Computational errors may occur due to failures at sites or communication link failures.

Local transaction manager at each site maintains the ACID properties of those transactions that execute at that site. Global transactions are executed with the cooperation of various transaction managers at different sites. Each site consists of two components: (a) the transaction manager that takes care of the execution of a local transaction or part of a global transaction that access data stored in the local site, (b) the transaction coordinator which will coordinate the execution of local and global transactions initiated at that site.

The transaction manager maintains a log for the purpose of recovery. It also participates in a concurrency control scheme so as to coordinate the concurrent execution of transactions at that site. The transaction coordinator is responsible for starting the execution of a transaction. It divides a global transaction into a number of subtransactions and distributes these transactions to the appropriate sites for execution. A transaction coordinator also coordinates the termination process of a transaction. In this process, the transaction may be committed at all sites or aborted at all sites.

2.4.1 Types of Failures in a Distributed Environment

In a distributed environment, apart from the same type of failures as in the centralized environment (for example, software errors, hardware errors or disk crashes), some additional types of failures are encountered: (a) failure of site, (b) loss of message, (c) failure of communication and (d) network partition. In distributed systems, the errors that results due to the loss or the corruption of messages are handled by using TCP/IP protocols. In the case of communication link failure,
messages that that would have been transmitted across the link must be rerouted. In some cases, another route through the network could be found to transmit the messages. In other cases, there can be failure because of no connection between some pairs of sites. A system is partitioned if it has been split into two or more subsystems, called partitions which lack any connection between them.

2.4.2 Commit Protocols

In order to ensure atomicity, a transaction must either commit at all sites or it must abort at all sites. A commit protocol is required to be executed by the transaction coordinator to ensure this property. Commonly used protocols are two phase commit protocol (2PC) [25, 26] and three phase commit protocol (3PC) [27].

Two Phase Commit Protocol

During normal execution, a log is maintained at each site for the purpose of logging the actions of a subtransaction. The transaction manager at the site where the transaction originated is called the coordinator for the transaction. Transaction managers at sites where its subtransactions execute are called subordinates. After the completion of transaction execution, at commit time, the commit command is sent to the coordinator for the transaction. This initiates the 2PC protocol [28].

1. The coordinator sends a prepare message to each subordinate.

2. After receiving a prepare message, the subordinate decides whether to commit or abort its subtransaction. It force-writes an abort or prepare log record, and sends a no or yes message to the coordinator.

3. After receiving yes message from all the subordinates, the coordinator force-writes a commit log record and then sends commit message to all subordinates.
In case, even one no message or no response is received by the coordinator from some subordinate for a certain time interval, the coordinator force-writes an abort log record, and then sends an abort message to all subordinates.

4. If an abort message is received by the subordinate, it force-writes an abort log record, sends ack message to the coordinator, and aborts the subtransaction. After the subordinate has received a commit message from the coordinator, it force-writes a commit log record, sends an ack message to the coordinator and commits the subtransaction.

5. When the coordinator receives ack messages from all the subordinates, it writes an end log record for the transaction.

Three Phase Commit Protocol

For the purpose of avoiding blocking problems even if the coordinator site fails during recovery, three phase commit (3PC) protocol was developed. Using this protocol, after the coordinator has sent prepare messages and received yes votes from all the subordinates, it sends a precommit message to all the sites, rather than a commit message. When the coordinator receives a sufficient number of acks which is more than the maximum number of failures that can be handled, it force-writes a commit log record and sends a commit message to all its subordinates. In this way, 3PC protocol allows the coordinator to postpone the decision to commit until it has ensured that sufficient number of sites know about the decision to commit. In case the coordinator fails subsequently, these sites can communicate with each other and detect that the transaction must be committed without waiting for the coordinator to recover. Conversely, when none of them has received a precommit message, the sites
can detect that the transaction must be aborted without waiting for the coordinator to recover.

2.4.3 Two Phase Locking Schemes for Distributed Database Systems

Two phase locking schemes which were developed for centralized database systems can be applied to serialize transactions in distributed database systems. However, some modifications are needed in the implementation in order to cater to the requirement of different data distribution model. The implementation of these schemes in distributed systems can be done in three ways: (1) Centralized two phase locking (2) primary copy locking (3) distributed two phase locking.

1. **Centralized two phase locking**: In this approach, for managing all locking activities, one node is made responsible. When a node requests for a lock, it is directed to the node where lock manager resides which makes the decision and informs the requesting node. In this approach, which is also known as Primary site two phase algorithm [29, 30], a transaction is processed at many sites. One of the sites is called the coordinating site and other sites are called participating sites for that transaction. Besides coordinating the transaction execution, the coordinating site sends a node’s locking request of a transaction to the central locking site. Locking service is only provided by the centralized locking site. If partial or full replication is required the coordinator selects the participating sites accordingly and completes updates for maintaining global consistency.

2. **Primary Copy Two Phase Locking Scheme**: The main disadvantage of the centralized two phase locking scheme is a single point of failure and other performance problems [31, 32]. These problems can be minimized if the
locking responsibility is distributed to many sites. Since each lock manager is responsible for a subset of data items, lock requests are sent to the appropriate lock manager by the node which is executing part of the transaction. In the case of full or partial replication, once a copy of the data item is locked, all copies of the data item are implicitly locked. In this way the locked copy of the data item serves as the primary copy of the data item.

3. Distributed Two Phase Locking Scheme: In this scheme, the locking responsibility is distributed to all nodes. Hence, all nodes have lock managers. The transaction coordinator sends lock request to all the participating sites which are responsible for managing the execution of their part of the transaction. As a result, instead of lock granted message, the participants are required to send only end of processing message to the coordinator. A distributed two phase locking scheme was implemented in System R* [33] and in NonStoop SQL [34, 35, 36].

2.5 Chapter Conclusions

In this chapter, the basic concepts of database systems and transactions in centralized and distributed environment were presented. The two phase locking schemes for centralized database systems are reviewed and the methods of applying this scheme to serialize transactions in distributed database systems are also discussed. The two phase commit protocol (2PC) and three phase commit protocol (3PC) have also been discussed. A detailed discussion on mobile database transactions is presented in the following chapter.