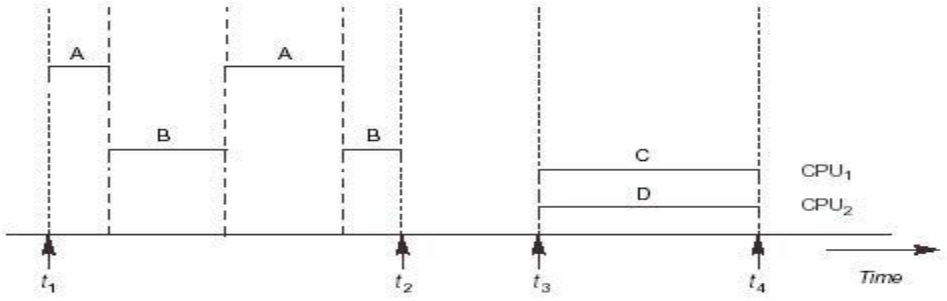
MODULE 5
Transaction Processing Concepts

5.1 Introduction to Transaction Processing

Single-User Versus Multiuser Systems

- A DBMS is **single-user** id at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.
- Most DBMS are multiuser (e.g., airline reservation system).
- *Multiprogramming operating systems* allow the computer to execute multiple programs (or processes) at the same time (having one CPU, concurrent execution of processes is actually interleaved).
- If the computer has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

Figure 19.1 Interleaved processing versus parallel processing of concurrent transactions.



5.2 Transactions, Read and Write Operations

- A *transaction* is a logical unit of database processing that includes one or more database access operations (e.g., insertion, deletion, modification, or retrieval operations). The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**

- *Read-only transaction* - do not changes the state of a database, only retrieves data.
- The basic database access operations that a transaction can include are as follows:

read_item(X): reads a database item *X* into a program variable *X*.

- *write_item(X):* writes the value of program variable *X* into the database item named *X*.

Executing a read_item(*X*) command includes the following steps:

3. Find the address of the disk block that contains item *X*.
4. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
5. Copy item *X* from the buffer to the program variable named *X*.

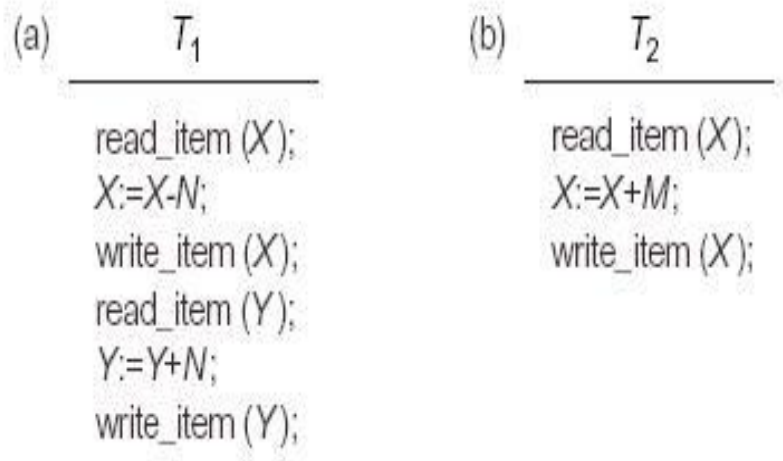
Executing a write_item(*X*) command includes the following steps:
Find the address of the disk block that contains item *X*.

6. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
7. Copy item *X* from the buffer to the program variable named *X*.

Executing a write_item(*X*) command includes the following steps:

6. Find the address of the disk block that contains item *X*.
7. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
8. Copy item *X* from the program variable named *X* into its correct location in the buffer.
9. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Figure 19.2 Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

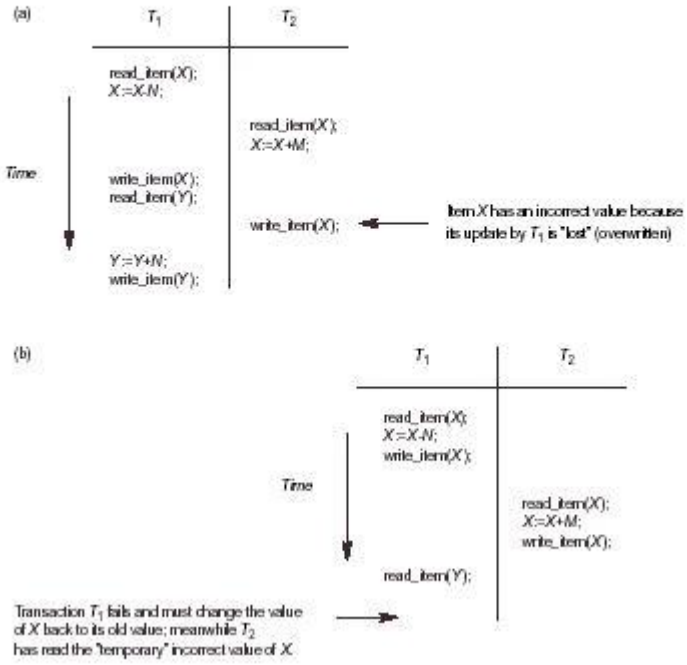


5.3 Why Concurrency Control Is Needed

- The Lost Update Problem.

 This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved then the final value of item *X* is incorrect, because T2 reads the value of *X before* T1 changes it in the database, and hence the updated value resulting from T1 is lost. For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ (T1 transfers 5 seat reservations from the flight corresponding to *X* to the flight corresponding to *Y*), and $M = 4$ (T2 reserves 4 seats on *X*), the final result should be $X = 79$; but in the interleaving of operations, it is $X = 84$ because the update in T1 that removed the five seats from *X* was *lost*

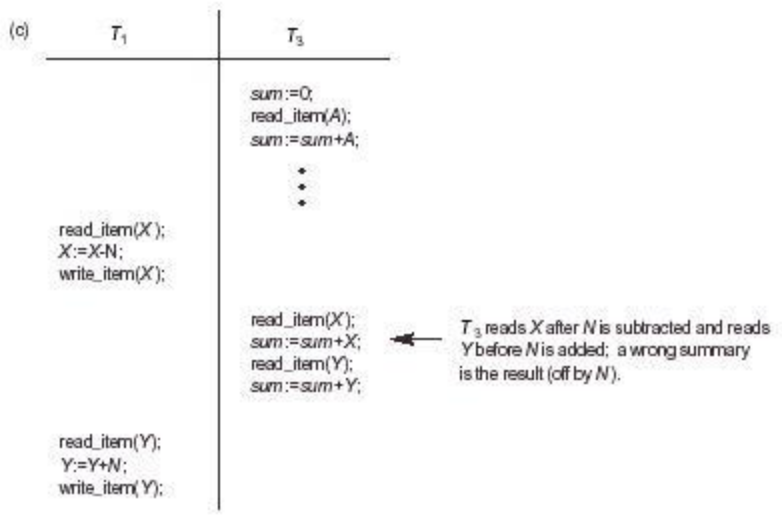
Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.



The Temporary Update (or Dirty Read) Problem.

This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value. Figure 19.03(b) shows an example where T1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction T2 reads the "temporary" value of *X*, which will not be recorded permanently in the database because of the failure of T1. The value of item *X* that is read by T2 is called *dirty data*, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem.*

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



- **The Incorrect Summary Problem.**

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure 19.03(c) occurs, the result of T3 will be off by an amount *N* because T3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it.

Another problem that may occur is called **unrepeatable read,** where a transaction *T* reads an item twice and the item is changed by another transaction *T'* between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

5.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) the transaction has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not. This may happen if a transaction **fails** after executing some of its operations but before executing all of them.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. *A computer failure (system crash):* A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. *A transaction or system error:* Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error . In addition, the user may interrupt the transaction during its execution.
3. *Local errors or exception conditions detected by the transaction:* During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition , such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
4. *Concurrency control enforcement:* The concurrency control method (see Chapter 20) may decide to abort the transaction, to be restarted later, because it violates serializability (see Section 19.5) or because several transactions are in a state of deadlock.
5. *Disk failure:* Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. *Physical problems and catastrophes:* This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

5.5 Transaction and System Concepts

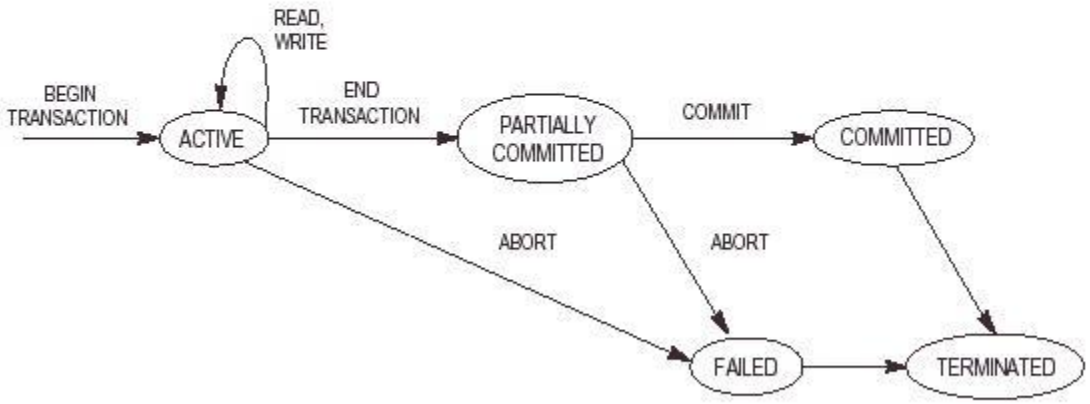
Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see below). Hence, the recovery manager keeps track of the following operations:

- BEGIN_TRANSACTION: This marks the beginning of transaction execution.
- READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.
- END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 19.5) or for some other reason.
- COMMIT_TRANSACTION: This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- ROLLBACK (or ABORT): This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone.*

Figure 19.04 shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log). Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state.** Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

Figure 19.4 State transition diagram illustrating the states for transaction execution.



5.6 The System Log

- To be able to recover from failures that affect transactions, the system maintains a *log* to keep track of all transactions that affect the values of database items.
- Log records consists of the following information (*T* refers to a unique *transaction_id*):
    1. [start_transaction, *T*]: Indicates that transaction *T* has started execution.
    2. [write_item, *T*,*X*,*old_value*,*new_value*]: Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value.*
    3. [read_item, *T*,*X*]: Indicates that transaction *T* has read the value of database item *X*.
    4. [commit,*T*]: Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    5. [abort,*T*]: Indicates that transaction *T* has been aborted.

5.7 Desirable Properties of Transactions

Transactions should posses the following (ACID) properties:

Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

5.8 Schedules and Recoverability

A **schedule** (or **history**) *S* of *n* transactions T1, T2, ..., Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in *S,* the operations of Ti in *S* must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in *S.* For now, consider the order of operations in *S* to be a *total ordering,* although it is possible theoretically to deal with schedules whose operations form *partial orders*.

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Similarly, the schedule for Figure 19.03(b), which we call Sb, can be written as follows, if we assume that transaction T1 aborted after its read_item(*Y*) operation:

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

1. they belong to different transactions;
2. they access the same item *X;* and
3. at least one of the operations is a write_item(*X*).

For example, in schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations w1(*X*) and w2(*X*). However, the operations r1(*X*) and r2(*X*) do not conflict, since they are both read operations; the operations w2(*X*) and w1(*Y*) do not conflict, because they operate on distinct data items *X* and *Y;* and the operations r1(*X*) and w1(*X*) do not conflict, because they belong to the same transaction.

A schedule *S* of *n* transactions T1, T2, ..., Tn, is said to be a **complete schedule** if the following conditions hold:

1. The operations in *S* are exactly those operations in T1, T2, ..., Tn, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

5.10 Characterizing Schedules Based on Recoverability

once a transaction *T* is committed, it should *never* be necessary to roll back *T*. The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called **nonrecoverable,** and hence should not be permitted.

A schedule *S* is recoverable if no transaction *T* in *S* commits until all transactions *T'* that have written an item that *T* reads have committed. A transaction *T* **reads** from transaction *T* in a schedule *S* if some item *X* is first written by T' and later read by *T*. In addition, T' should not have been aborted before *T* reads item *X*, and there should be no transactions that write *X* after T' writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *X).*

Consider the schedule S'_a given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

$$S'_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$$

S_a' is recoverable, even though it suffers from the lost update problem. However, consider the two (partial) schedules S_c and S_d that follow:

$$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$$
$$S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$$
$$S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$$

)

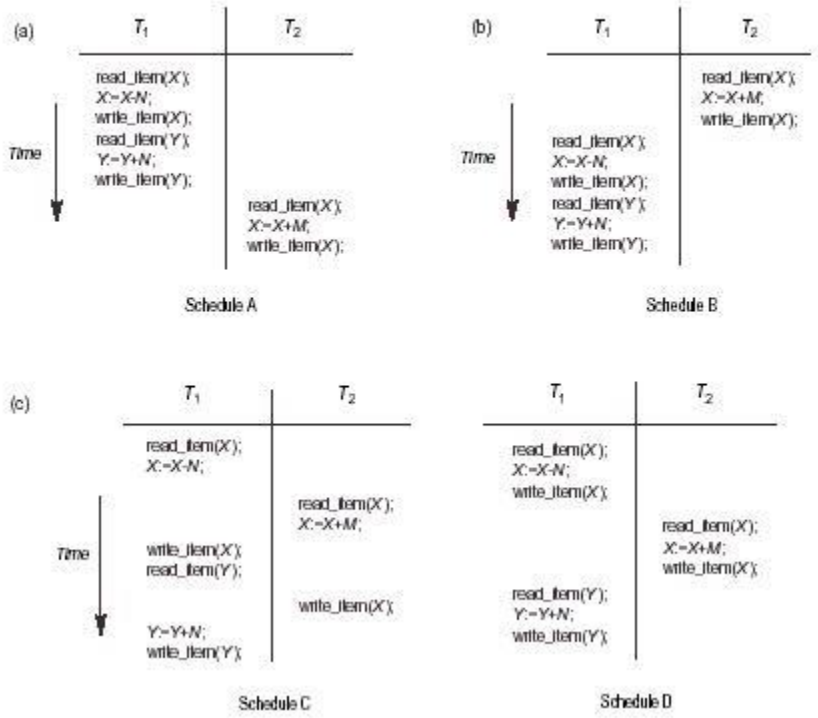
S_e is not recoverable, because T2 reads item *X* from T1, and then T2 commits before T1 commits. If T1 aborts after the c2 operation in S_e, then the value of *X* that T2 read is no longer valid and T2 must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c2 operation in S_e must be postponed until
after T1 commits. If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of *X* it read is no longer valid.

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

Serializability of Schedules

- If no interleaving of operations is permitted, there are only two possible arrangement for transactions T1 and T2.
    1. Execute all the operations of T1 (in sequence) followed by all the operations of T2 (in sequence).
    2. Execute all the operations of T2 (in sequence) followed by all the operations of T1
- A schedule *S* is *serial* if, for every transaction *T* all the operations of *T* are executed consecutively in the schedule.
- A schedule *S* of *n* transactions is *serializable* if it is equivalent to some serial schedule of the same *n* transactions.

Figure 19.5 Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



5.11 Transaction Support in SQL

- An SQL transaction is a logical unit of work (i.e., a single SQL statement).
- The *access mode* can be specified as *READ ONLY* or *READ WRITE*. The default is *READ WRITE*, which allows update, insert, delete, and create commands to be executed.
- The *diagnostic area size* option specifies an integer value *n*, indicating the number of conditions that can be held simultaneously in the diagnostic area.
- The *isolation level* option is specified using the statement *ISOLATION LEVEL*.
- the default isolation level is *SERIALIZABLE*.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
  READ WRITE
  DIAGNOSTICS SIZE 5
  ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
  VALUES ('Jabbar', 'Ahmad', '998877665', 2, 44000);
EXEC SQL UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
 GOTO THE_END;
UNDO:  EXEC SQL ROLLBACK;
THE_END: . . . ;
```

Questions

1. Write a short Notes on

    i. 2PL Lock
    ii. Two-P Deadlock

2. Three phase Locking Techniques: Essential components
3. Explain properties of a transaction with state transition diagram.
4. What is a schedule? Explain with example serial, non serial and conflict serializable schedules.
5. Write short notes on
    1. Write ahead log protocol
    2. Time stamp Ordering
    3. Two phase locking protocol
6. Explain the problems that can occur whaen concurrent transaction are executed give examples