Chapter 7

Transaction Processing

7.1 INTRODUCTION

A transaction is a discrete unit of work that must be completely processed or not processed at all.

Example: transferring funds from a checking account to a saving account.

A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes. The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions. Transaction processing divides information process into individual, indivisible operations, called transactions that complete or fail as a whole; a transaction can't remain in an intermediate, incomplete, state. So other processes can't access the transaction's data until either the transaction has completed or it has been "rolled back" after failure.

Examples includes: systems for reservations, banking, credit card processing, stock markets, supermarket checkout, and other similar systems. They require high availability and fast response time for hundreds of concurrent users.

Transaction processing is designed to maintain database integrity of related data items in a consistent state. Transaction processing databases are databases that have been designed specifically to optimize the performance of transaction processing, which is often referred to as OLTP (Online Transaction Processing).

7.1.1 Transaction Concepts

A transaction is a logical unit of work. It begins with the execution of a BEGIN TRANSACTION operation and ends with the execution of a COMMIT or ROLLBACK operation.

A Sample Transaction (Pseudo code)

```
BEGIN TRANSACTION
        UPDATE ACC123 (BALANCE: =BALANCE-$100);
                If any error occurred THEN GOTO UNDO;
                END IF;
```

```
          UPDATE ACC456 (BALANCE: =BALANCE+$100);
               If any error occurred THEN GOTO UNDO;
               END IF;
     COMMIT;
               GOTO FINISH;
     UNDO;
               ROLLBACK;
     FINISH;
               RETURN;
```

In our example an amount of $100 is transferred from account 123 to 456. It is not a single atomic operation; it involves two separate updates on the database. Transaction involves a sequence of database update operation. The purpose of this transaction is to transform a correct state of database into another correct state, without preserving correctness at all intermediate points.

Transaction management guarantees a correct transaction and maintains the database in a correct state. It guarantees that if the transaction executes some updates and then a failure occurs before the transaction reaches its planned termination, then those updates will be undone. Thus the transaction either executes entirely or totally cancelled. The system component that provides this atomicity is called transaction manager or transaction processing monitor or TP monitor. ROLLBACK and COMMIT are keys to the way it works.

1. **COMMIT**
   - The COMMIT operation signals successful end of transaction.
   - It tells the transaction manager that a logical unit of work has been successfully completed and database is in correct state and the updates can be recorded or saved.
2. **ROLLBACK**
   - By contrast, the ROLLBACK operation signals unsuccessful end of transaction.
   - It tells the transaction manager that something has gone wrong, the database might be in incorrect state and all the updates made by the transaction should be undone.
3. **IMPLICIT ROLLBACK**
   - Explicit ROLLBACK cannot be issued in all cases of transaction failures or errors. So the system issues implicit ROLLBACK for any transaction failure.
   - If the transaction does not reach the planned termination then we ROLLBACK the transaction else it is COMMITTED.
4. **MESSAGE HANDLING**
   - A typical transaction will not only update the database, it will also send some kind of message back to the end user indicating what has happened.

     Example: "Transfer done" if the COMMIT is reached, or "Error—transfer not done"
5. **RECOVERY LOG**
   - The system maintains a log or journal or disk on which all particular about the updation is maintained.

- The values of before and after updation is also called as before and after images.
- This log is used to bring the database to the previous state in case of some undo operation.
- The log consist of two portions
  - an *active* or online portion
  - an *archive* or offline portion.
- The **online portion** is the portion used during normal system operation to record details of updates as they are performed and it is normally kept on disk.
- When the online portion becomes full, its contents are transferred to the **offline portion**, which can be kept on tape.

6. STATEMENT ATOMICITY

- The system should guarantee that individual statement execution must be atomic.

7. PROGRAM EXECUTION

- Program execution is a sequence of transactions.
- COMMIT and ROLLBACK terminate the *transaction,* not the application program.
- A single program execution will consist of a *sequence* of several transactions running one after another. A single program execution will consist of a sequence of several transactions running one after another as illustrated below

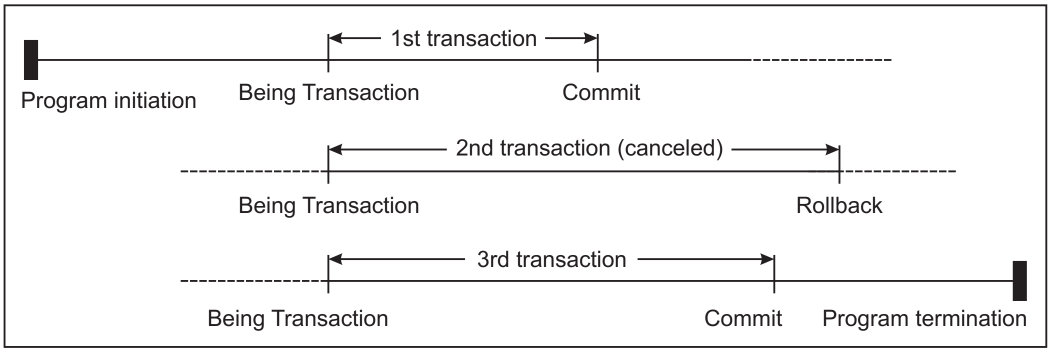

Fig. 7.1. *Program execution is a sequence of transactions.*

8. NO NESTED TRANSACTIONS

- An application program can execute a BEGIN TRANSACTION statement only when it has no transaction currently in progress.
- i.e., no transaction has other transactions nested inside itself.

9. CORRECTNESS

- Consistent means "not violating any known integrity constraint."
- Consistency and correctness of the system should be maintained.
- If *T* is a transaction that transforms the database from state *D1* to state *D2,* and if *D1* is correct, then *D2* is correct as well.

10. MULTIPLE ASSIGNMENT

- Multiple assignments allow any number of individual assignments (i.e., updates) to be performed "simultaneously."

  Example: UPDATE ACC 123 {BALANCE: = BALANCE - $100}

  UPDATE ACC 456 {BALANCE: = BALANCE + $100}
- Multiple assignments would make the statement atomic.
- Current products do not support multiple assignments.

7.2 REASONS FOR TRANSACTION FAILURES

Failures can be 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. 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 some cases, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction:** During transaction execution there may be occurrence of certain condition which may require the cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exceptional 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 may decide to abort the transaction, to be restarted later, because it violates serializability 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 disaster:** 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.

7.3 TRANSACTION AND SYSTEM CONCEPTS

A transaction is an atomic unit of work that is either completed entirey or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. The recovery manager is responsible to keep track of those details.

7.3.1 Transaction States and Additional 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 or whether the transaction has to be aborted because it violates serializability or for some other reason.
- **COMMIT_TRANSACTION:** This signals a successful end of the transaction so that any changes 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.

7.3.2 State Transition of a State Transition Diagram

Whenever a transaction is submitted to a DBMS for execution, either it executes successfully or fails due to some reasons. During its execution, a transaction passes through various states that are active, partially committed, committed, failed, and aborted. The following figure shows the state transition diagram that describes how a transaction passes through various states during its execution.

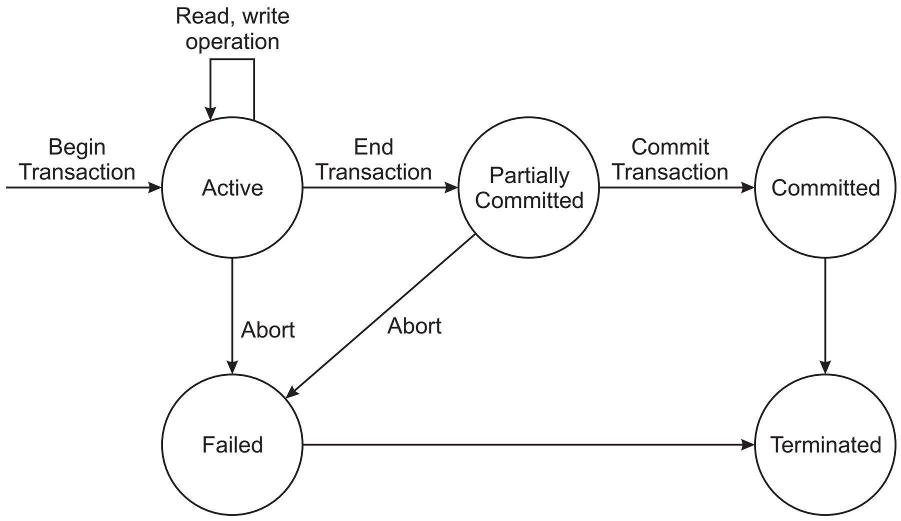

Fig. 7.2. *State transition diagram of a transaction.*

State transition diagram 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 does not occur. 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, the execution is completed successfully and all its changes can be recorded permanently in the database.

A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later, either automatically or after being resubmitted by the user.

7.4 THE SYSTEM LOG

To recover the system from transaction failure, the system maintains a log. This log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from failures. The log is kept on disk, so it is not affected by any type of failure except for disk or disastrous failure.

The log is periodically backed up to archival storage like tape to guard against such disastrous failures. The entry made in such log is called log records. The log records and the action performed are as follows:

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 to the database.
5. **[abort, *T*]:** Indicates that transaction *T* has been aborted.

T refers to a unique transaction-id that is generated automatically by the system and is used to uniquely identify each transaction.

All READ operations are not required to be written to the system log. However, if the log is also used for other purposes like keeping track of all database operations then such entries can be included.

If the system crashes, we can recover to a consistent database state by examining the log. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to undo the effect of these WRITE operations of a transaction *T* by tracing backward through the log and resetting all items changed by a WRITE operation of *T* to their old_values. Redoing the operations of a transaction may also be needed if all its updates are recorded in the log but a failure occurs before we can be sure that all these new_values have been written permanently in the actual database. Redoing the operations of transaction *T* is applied by tracing forward through the log and setting all items changed by a WRITE operation of *T* to their new_values.

7.5. ACID PROPERTIES OR TRANSACTION PROPERTIES

Transactions possess several properties and these properties are often called as the ACID properties. These properties are responsible for enforcing 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 entirey or not performed at all.

   The atomicity property requires the complete execution of the transaction. 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.

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take the database from one consistent state to another.

   The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. State of a database is a collection of all the stored data items in the database at a given point in time. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A database program should be written in a way that guarantees the consistent state of the database before and after the execution of the transaction, assuming that there is no occurrence of interference with other transactions.

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.

   Isolation is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks. There have been attempts to define the *level of isolation* of a transaction. A transaction is said to have level 0 isolation if it does not overwrite the dirty reads of higher-level transactions. A level 1 isolation transaction has no lost updates; and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation or true isolation has no lost updates, no dirty reads and repeatable reads.

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.

   Finally, the durability property is the responsibility of the recovery subsystem of the DBMS.

7.6 SERIALIZABILITY

Serializability is the generally accepted "criterion for correctness" for the interleaved execution of a set of transactions. Interleaved execution is considered to be correct if and only if it is serializable. The execution of a given set of transactions is serializable and correct if and only if it is equivalent to some serial execution of the same transactions, where:

- A serial execution is one in which the transactions run one at a time in some sequence.
- Guaranteed means that the given serial execution of the transaction always produces the same result as each other, no matter what the initial state of the database might be.

  The definition is justified as follows:

  1. Individual transactions are assumed to be correct: i.e., they are assumed to transform a correct state of the database into another correct state.
  2. Running the transactions one at a time in any serial order is correct ("any" serial order is correct because individual transactions are assumed to be independent of one another).
  3. It is thus reasonable to define an interleaved execution to be correct if and only if it is equivalent to some serial execution.
  4. Some times nonserializable interleaved execution of a transaction may also lead to correct result in some specific initial state of the database.
- It is found in the examples of lost update problem, uncommitted dependency problem and inconsistent analysis problem (discussed in next chapter) that interleaved execution was not serializable.
- The use of strict two phase locking protocol forced serializable but it led to another problem called deadlock.

Given a set of transactions, any execution of those transactions, interleaved or otherwise, is called a schedule. Executing the transactions one at a time, with no interleaving, constitutes a serial schedule. Two schedules are said to be equivalent if and only if they produce the same result, no matter about the initial state of the database.

Two different serial schedules involving the same transactions will produce different results and hence the two different interleaved schedules involving those same transactions might also produce different results, and yet both are correct.

For example, suppose transaction A is of the form “Add 1 to x” and transaction B is of the form “Double x”. Suppose that the initial value of x is 10. Then the serial schedule A then B gives x=22, the serial schedule B then A gives x=21.

The two phase locking theorem states that *“if all transactions obey two phase protocol, then all possible interleaved schedules are serializable”*. The two phase locking protocol is as follows.

- Before operating on any object (e.g., a database tuple), a transaction must acquire a lock on that object.
- After releasing a lock, a transaction must never go on to acquire any more locks.

A transaction that obeys this protocol has two phases, a lock acquisition or "growing" phase and a lock releasing or "shrinking" phase. Shrinking phase is compressed into the single operation of COMMIT or RDLLBACK at end-of-transaction. Let I be an interleaved schedule involving some set of transactions T1, T2,…,Tn. If I is serializable, then there exists at least one serial-schedule S involving T1,T2, ...,Tn such that I is equivalent to S. S is said to be a serialization of I.

If A and B are any two transactions involved in some serializable schedule, then either A logically precedes B or B logically precedes A in that schedule. Conversely if the effect is not as if either A ran before B or B ran before A, then the schedule is not serializable and not correct.

In order to reduce resource utilization and thereby improving performance and throughput, real-world systems allows the construction of transactions that are not two-phase i.e., transactions "release locks early" (prior to COMMIT) and then acquire more locks. However, such transactions are a risky.

7.6.1 Schedules and Serializability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule or history. The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before a task is assigned to the database, first thing to be decided is the schedule that maintains the database in consistent state.

Transactions are programs; it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. Let us consider two operations: read and write. Between a read(*Q*) instruction and a write(*Q*) instruction on a data item *Q*, a transaction may perform an random sequence of operations on the copy of *Q* that is residing in the local buffer of the transaction. Therefore only read and write instructions are shown in any schedules, as shown in the figure below.

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Fig. 7.3. *Schedule 1: Schedule showing read and write operations.*

7.6.2 CONFLICT SERIALIZABILITY

Let us consider a schedule *S* in which there are two consecutive instructions I_i and I_j, of transactions T_i and T_j, respectively ($i \neq j$). If *Ii* and *Ij* refer to different data items, then we can swap I_i and I_j without affecting the results of any instruction in the schedule. However, if I_i and I_j refer to the same data item *Q*, then the order of the two steps may matter. There are four cases to consider:

1. I_i = read(Q), I_j = read(Q). The order of I_i and I_j does not matter, since the same value of Q is read by *Ti* and *Tj*, regardless of the order.
2. I_i = read(Q), I_j = write(Q). If I_i comes before I_j, then *Ti* does not read the value of Q that is written by T_j in instruction I_j. If I_j comes before I_i, then T_i reads the value of Q that is written by T_j. Thus, the order of I_i and I_j matters.
3. I_i = write(Q), I_j = read(Q). The order of I_i and I_j matters for reasons similar to those of the previous case.
4. I_i = write(Q), I_j = write(Q). Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database.

If there is no other write(Q) instruction after I_i and I_j in S, then the order of I_i and I_j directly affects the final value of Q in the database state that results from schedule S. Thus, only in the case where both I_i and I_j are read, instructions do the relative order of their execution, not matter. We say that I_i and I_j conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule in Fig 7.3. The write (A) instruction of T_1 conflicts with the read (A) instruction of T_2. However, the write (A) instruction of T_2 does not conflict with the read (B) instruction of T_1, because the two instructions access different data items. Let I_i and I_j be consecutive instructions of a schedule S.

If I_i and I_j are instructions of different transactions and I_i and I_j do not conflict, then we can swap the order of I_i and I_j to produce a new schedule S^1. We expect S to be equivalent to S^1, since all instructions appear in the same order in both schedules except for I_i and I_j, whose order does not matter. Since the write (A) instruction of T_2 in schedule of Figure 7.3 does not conflict with the read (B) instruction of T_1, we can swap these instructions to generate an equivalent schedule as in figure 7.4. Regardless of the initial system state, both schedules (1 and 2) produce the same final system state. We continue to swap non conflicting instructions:

- Swap the read (B) instruction of *T*1 with the read (A) instruction of *T*2.
- Swap the write (B) instruction of *T*1 with the write (A) instruction of *T*2.
- Swap the write (B) instruction of *T*1 with the read (A) instruction of *T*2.

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

Fig. 7.3. *Schedule 2: Schedule showing read and write operations after swapping.*

The final result of these swaps is a serial schedule and it's shown in Fig. 7.4. Schedule 3. This equivalence implies that, regardless of the initial system state, schedule 1 will produce the same final state as some other serial schedule. If a schedule *S* can be transformed into a schedule S^1 by a series of swaps of non conflicting instructions, then *S* and S^1 are conflict equivalent.

T_1	T_2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Fig. 7.4. *Schedule 3: A serial schedule that is equivalent to schedule 1.*

7.6.3 View Serializability

Consider two schedules *S* and S^1, where the same set of transactions participates in both schedules. The schedules *S* and S^1 are said to be view equivalent if three conditions are met:

1. For each data item *Q*, if transaction T_i reads the initial value of *Q* in schedule *S*, then transaction T_i must, in schedule S^1, also read the initial value of *Q*.
2. For each data item *Q*, if transaction T_i executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction T_j , then the read(*Q*) operation of transaction T_i must, in schedule S^1, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction T_j .
3. For each data item *Q*, the transaction (if any) that performs the final write(*Q*) operation in schedule *S* must perform the final write(*Q*) operation in schedule S^1.

The concept of view equivalence leads to the concept of view serializability. Schedule *S* is said to be view serializable if it is view equivalent to a serial schedule. Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. The schedule in figure 7.5 is view-serializable but not conflict serializable

T3	T4	T6
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)

Fig. 7.5. *Schedule 4: View Serializability.*

7.6.4 Testing for Serializability

Algorithm can be used to test a schedule for conflict serializability. The algorithm looks at only the read and writes item operations in a schedule to construct a precedence graph. The precedence graph is a directed graph G=(N,E) that consists of set of nodes N={T_1, T_2,…,T_n} and a set of directed edges E={e_1,e_2,…,e_m}. For each transaction in the schedule there is one node in the graph.

Each edge e_i in the graph is of the form ($T_j \rightarrow T_k$) where T_j is the starting node of e_i and T_k is the ending node of e_i. Such an edge is created if one of the operations in T_j appears in the schedule before some conflicting operation in T_k.

Algorithm

1. For each transaction T_i participating in schedule S, create a node labeled in the precedence graph.
2. For each case in S where T_j executes a read item(x) after T_i, executes a write-item(x), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write-item(x) after T_i executes a read-item(x) create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write-item(x) after T_i executes a write-item(x) create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule is serializable if and only if the precedence graph has no cycle.

REVIEW QUESTIONS

1. Explain the concept of transaction.
2. Explain the different states of a transaction with a neat diagram.
3. Explain ACID properties.
4. What is the need for system log in a transaction management system?
5. Explain serializability.
6. Explain the types of serializability with an example schedule.
7. Illustrate the algorithm to test for serializability.

❑❑❑