Transaction

A transaction is an action, or series of actions that are being performed by a single user or application program, which reads or updates the contents of the database.

A transaction can be defined as a logical unit of work on the database. This may be an entire program, a piece of a program or a single command (like the SQL commands such as INSERT or UPDATE) and it may engage in any number of operations on the database.

Transaction Properties (Features)

In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called ACID properties.

Atomicity - This property states that each transaction must be considered as a single unit and must be completed fully or not completed at all. No transaction in the database is left half completed. Database should be in a state either before the transaction execution or after the transaction execution. It should not be in a state 'executing'.

Consistency - Any transaction should not inject any incorrect or unwanted data into the database. A transaction must preserve the consistency of a database after the execution.

Isolation - If there are multiple transactions executing simultaneously, then all the transaction should be processed as if they are single transaction. But individual transaction in it should not alter or affect the other transaction. The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. Transactions occur independently without interference.

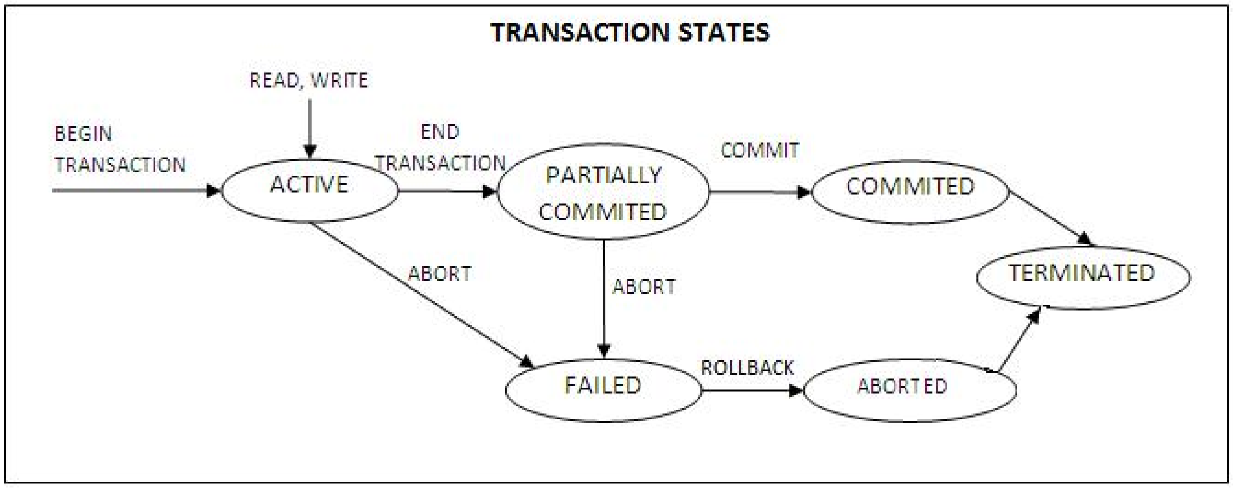
Durability - The database should be durable enough to hold all its latest updates even if the system fails or restarts. It should not be working for single transaction alone. It should be able to handle multiple transactions too.

States of Transaction

A transaction must be in one of the following states:

1. **Active**: It is the initial state in which the transaction is executing.
2. **Partially committed**: It is the state after the final statement has been executed.
3. **Failed**: The transaction is said to be failed when the normal execution can no longer proceed.
4. **Aborted**: When the transaction is failed then after the transaction is rolled back and the database is restored to its state prior to the start of the transaction. The transaction in this state is said to be aborted.

5. **Committed**: It is the state after the successful completion of the transaction.
6. **Terminated**: It is the state in which the transaction finally reaches, whether it is committed or aborted.



Serializability

Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations.

Recoverability

Recoverability refers to the ability to restore your deployment to the point at which a failure occurred. The ability to recover quickly from a system failure or disaster depends not only on having current backups of your data, but also on having a predefined plan for recovering that data on new hardware.

Concurrency control

Concurrency control is the process of managing simultaneous execution of transactions (such as queries, updates, inserts, deletes and so on) in a multiprocessing database system without having them interfere with one another.

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories −

- **Lock based protocols**
- **Time stamp based protocols**

<u>Lock-based Protocols</u>

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

<u>Binary Locks</u> − A lock on a data item can be in two states; it is either locked or unlocked.

<u>Shared/exclusive</u> − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

<u>Timestamp-based Protocols</u>

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last 'read and write' operation was performed on the data item.

<u>Validation Based Protocol</u>

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in following three phases:

<u>Read phase</u>: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without update to the actual database.

<u>Validation phase</u>: In this phase, the temporary variable value will bevalidated against the actual data to see if it violates the serializability.

Write phase: If the validation of the transaction is validated then the temporary results are written to the database or system otherwise the transaction is rolled back.

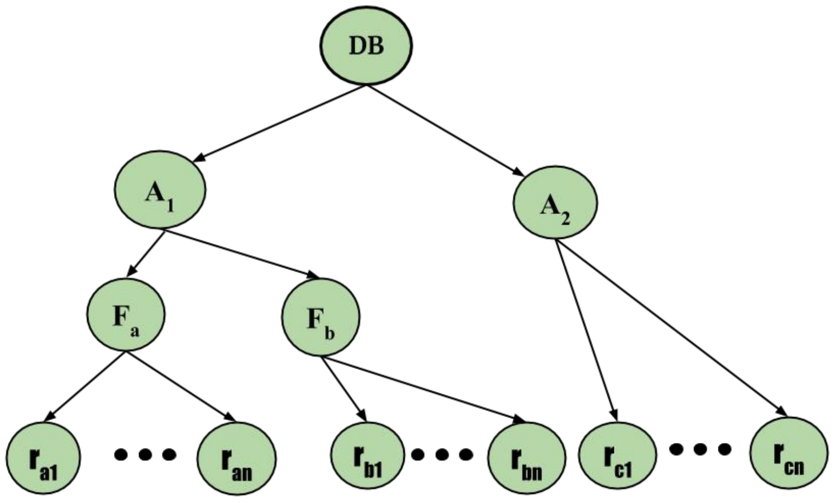
Multiple Granularity

It can be defined as hierarchically breaking up the database into blocks which can be locked.

This Multiple Granularity protocol enhances concurrency and reduces lock overhead.

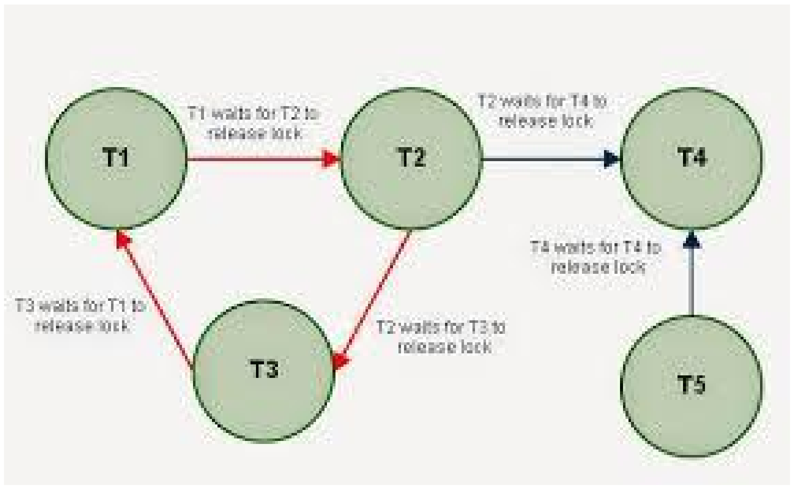
It maintains the track of what to lock and how to lock.

It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.



Deadlocks

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction.


T1 waits for T2 to release lock
T2 waits for T4 to release lock
T1
T2
T4
T4 waits for T4 to release lock
T3 waits for T1 to release lock
T2 waits for T3 to release lock
T3
T5

Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

There are two algorithms for this purpose, namely wait-die and wound-wait. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

Wait-Die − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.

Wound-Wait − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.