DBMS

UNIT-IV

Transaction Concept

A transaction is a unit of program execution that accesses and possibly updates various data items.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

1.Atomicity - Either all operations of the transaction are reflected properly in the database, or none are.

2.Consistency - Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

3.Isolation - Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

4.Durability - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

In the absence of failures, all transactions complete successfully. However, as noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed aborted. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database.

Following are the states of a transaction:

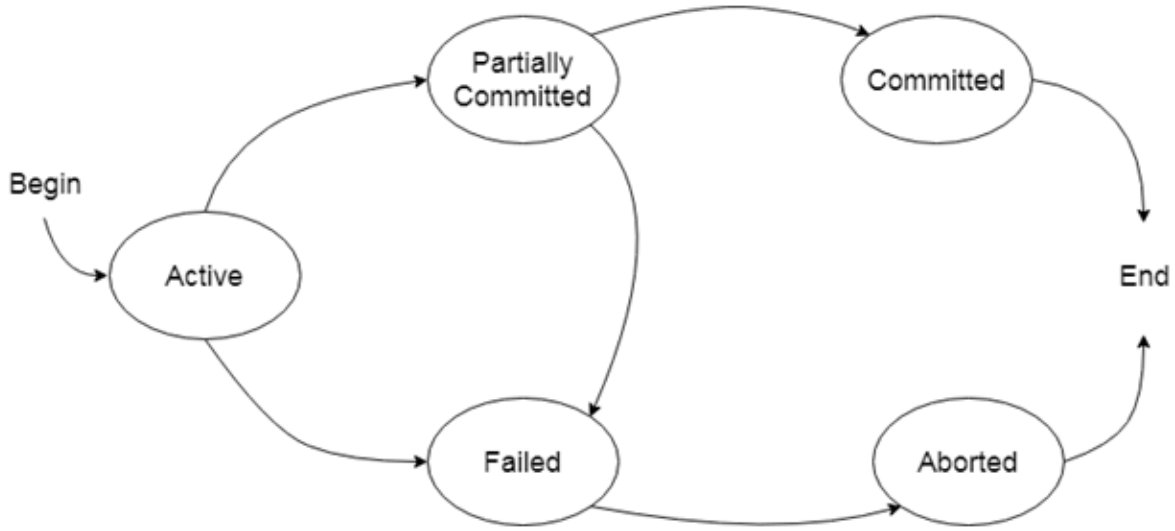
1.Active - the initial state; the transaction stays in this state while it is executing

2.Partially committed - after the final statement has been executed

3.Failed - after the discovery that normal execution can no longer proceed

4.Aborted - after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

5.Committed - after successful completion.



State diagram of a transaction

Concurrency Control

One of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called concurrency-control schemes. The concurrency-control schemes that we discuss in this chapter are all based on the serializability property. That is, all the schemes presented here ensure that the schedules are serializable.

Lock-Based Protocols One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Locks - There are various modes in which a data item may be locked.

1. Shared - If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.

2. Exclusive - If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

	S	**X**
S	True	False
X	False	True

Lock-compatibility matrix comp

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario.

Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock. Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish. But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved. We can avoid starvation of transactions by granting locks in the following manner: When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that

1.There is no other transaction holding a lock on Q in a mode that conflicts with M.

2.There is no other transaction that is waiting for a lock on Q, and that made its lock request before Ti.

Thus, a lock request will never get blocked by a lock request that is made later.

The Two-Phase Locking Protocol

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase -** A transaction may obtain locks, but may not release any lock.

2. **Shrinking phase** - A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

	T5	T6	T7

lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Partial schedule under two-phase locking

Timestamps

With each transaction Ti in the system, we associate a unique fixed timestamp, de-noted by TS (Ti). This timestamp is assigned by the database system before the trans-action Ti starts execution. If a transaction Ti has been assigned timestamp TS (Ti), and a new transaction Tj enters the system, then TS (Ti) <TS (Tj). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's time-stamp is equal to the value of the clock when the transaction enters the system.

2.Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The timestamps of the transactions determine the serializability order.

Thus, if TS (Ti) <TS (Tj), then the system must ensure that the produced schedule is equiva-lent to a serial schedule in which transaction Ti appears before transaction Tj. To implement this scheme, we associate with each data item Q two timestamp values:

•**W-timestamp** (Q) denotes the largest timestamp of any transaction that executed write (Q) successfully.

•**R-timestamp** (Q) denotes the largest timestamp of any transaction that executed read (Q) successfully.
These timestamps are updated whenever a new read (Q) or write (Q) instruction is executed.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.

- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

    - If W_TS(X) >TS (Ti) then the operation is rejected.
    - If W_TS(X) <= TS (Ti) then the operation is executed.
    - Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write (X)** operation:

- If TS (Ti) < R_TS(X) then the operation is rejected.
- If TS (Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.