BCA204T: DATA BASE MANAGEMENT SYSTEMS

Unit - V

Transaction Processing Concepts: Introduction, Transaction and System Concepts, Desirable properties of transaction, Schedules and Recoverability, Serializability of Schedules, Transaction Support in SQL, Locking Techniques for Concurrency Control, Concurrency Control based on time stamp ordering.

Unit-V

Transaction Processing Concepts

A **Transaction** is a Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).

Or

A transaction can be defined as a group of tasks. A single task is the minimum processing unit of work, which cannot be divided further.

An example of transaction can be bank accounts of two users, say A & B. When a bank employee transfers amount of Rs. 500 from A's account to B's account, a number of tasks are executed behind the screen. This very simple and small transaction includes several steps: decrease A's bank account from 500

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

In simple words, the transaction involves many tasks, such as opening the account of A, reading the old balance, decreasing the 500 from it, saving new balance to account of A and finally closing it. To add amount 500 in B's account same sort of tasks need to be done:

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

A simple transaction of moving an amount of 500 from A to B involves many low level tasks.

ACID Properties of a Transaction

The execution of a transaction Ti must satisfy some properties that are referred to as „ACID". The acronym "**ACID**" stands for „**A**tomicity", „**C**onsistency", „**I**solation" and „**D**urability", which are explained below:-

Atomicity

Though a transaction involves several low level operations but this property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in database where the transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

Consistency

This property states that after the transaction is finished, its database must remain in a consistent state. There must not be any possibility that some data is incorrectly affected by the execution of transaction. If the database was in a consistent state before the execution of the transaction, it must remain in consistent state after the execution of the transaction.

Isolation

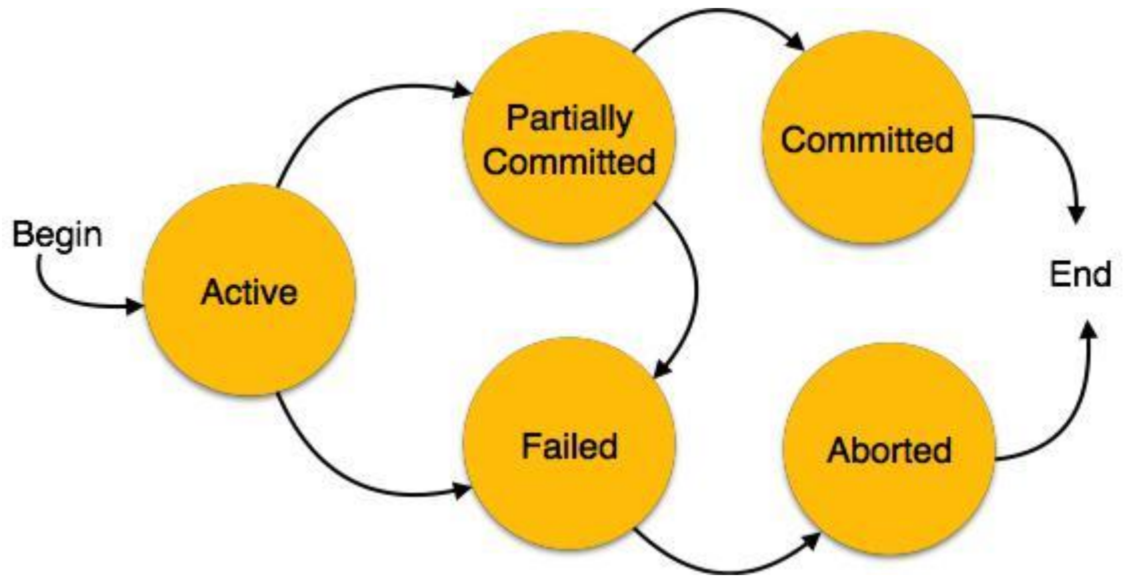
In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Durability

This property states that in any case all updates made on the database will persist even if the system fails and restarts. If a transaction writes or updates some data in database and commits that data will always be there in the database. If the transaction commits but data is not written on the disk and the system fails, that data will be updated once the system comes up.

States of Transactions:

A transaction in a database can be in one of the following state:



Active: In this state the transaction is being executed. This is the initial state of every transaction.

Partially Committed: When a transaction executes its final operation, it is said to be in this state. After execution of all operations, the database system performs some checks e.g. the consistency state of database after applying output of transaction onto the database.

Failed: If any checks made by database recovery system fails, the transaction is said to be in failed state, from where it can no longer proceed further.

Aborted: If any of checks fails and transaction reached in Failed state, the recovery manager rolls back all its write operation on the database to make database in the state where it was prior to start of execution of transaction. Transactions in this state are called aborted. Database recovery module can select one of the two operations after a transaction aborts:

- Re-start the transaction
- Kill the transaction

Committed: If transaction executes all its operations successfully it is said to be committed. All its effects are now permanently made on database system.

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:

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.

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.

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.

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.

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.

Physical problems and catastrophes*:* This refers to an endless list of problems (usually sudden damage) 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.

LOCKING TECHNIQUES FOR CONCURRENCY CONTROL

A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Types of Locks and System Lock Tables

Several types of locks are used in concurrency control.

Binary Locks : A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item 'X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested. We refer to the current value (or state) of the lock associated with item X as LOCK(X).

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an un1ock_i tem(X) operation, which sets LOCK(X) to 0(unlocks the item) so that 'X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation 1ock_i tem(X) before any read_i tem(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_i tem(X) after all read_i tem(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_i tem(X) operation if it already holds the lock on item X.

4. A transaction T will not issue an unlock_i tern*(X)* operation unless it already holds the lock on item X.

Shared/Exclusive (or Read/Write) Locks: The preceding binary locking scheme is too restrictive for database items, because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock called a **multiple mode lock** is used. In this scheme-called **shared/exclusive** or **read/write locks** there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: "read-locked," "writelocked," or "unlocked." A **readlocked item** is also called **share-locked**, because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked**, because a single transaction exclusively holds the lock on the item.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or wri te_l ock(X) before any read_i tem(X) operation is performed in T.
2. A transaction T must issue the operation wri te_l ock(X) before any wri te_i tem(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_i tem(X) and wri te_i tem(X) operations are completed in T.
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction T will not issue a wri te_l ock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

The two Phase locking(2PL) protocol

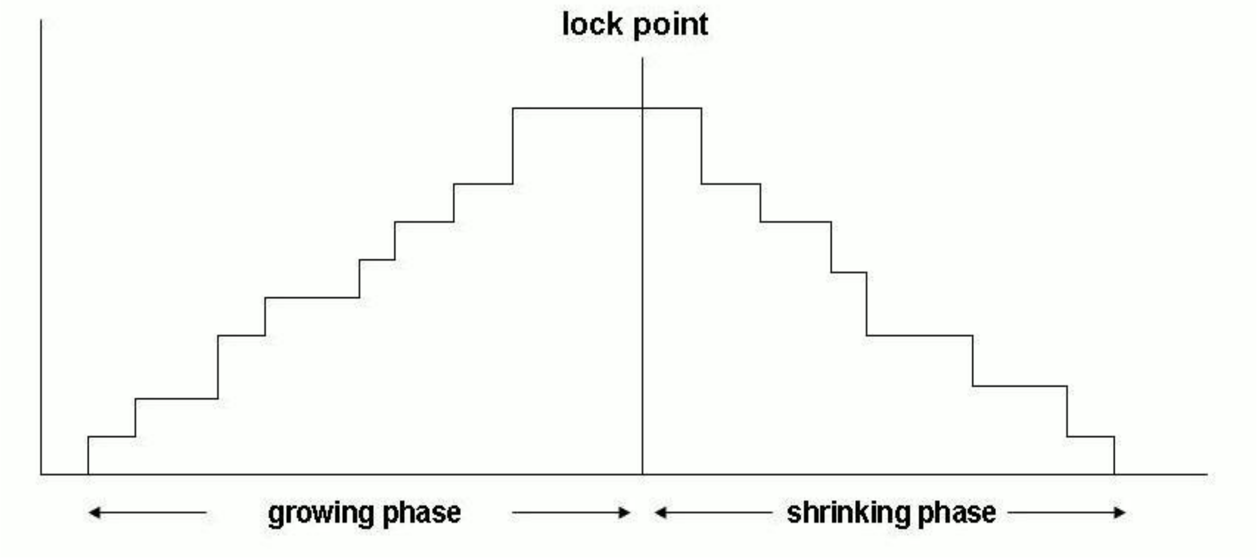
The two-phase locking protocol is used to ensure the serializability in Database. This protocol is implemented in two phase

- **Growing Phase**

  In this phase we put read or write lock based on need on the data. In this phase we does not release any lock.

- **Shrinking Phase**

  This phase is just reverse of growing phase. In this phase we release read and write lock but doesn't put any lock on data.



CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamp of transaction T as TS(T).

Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called timestamp ordering (TO).

The timestamp algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. Read_TS(X): The read timestamp of item Xi this is the largest timestamp among all the timestamps of
transactions that have successfully read item X-that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully.
2. Write_TS(X): The write timestamp of item Xi this is the largest of all the timestamps of transactions that have successfully written item X-that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully.

Whenever some transaction T times to issue a read item(X) or a write_item(X) operation, the basic TO algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp.* If T is aborted and rolled back, any transaction T 1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.

We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a write_item(X) operation:
   a) If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)-and hence *after* T in the timestamp ordering-has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b) If the condition in part (a) does not occur, then execute the wri te_i tem(X) operation ofT and set write_TS(X) to TS(T).
2. Transaction T issues a read_item(X) operation:
   a) If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)-and hence *after* T in the timestamp ordering-has already written the value of item X before T had a chance to read X.
   b) If write_TS(X) ≤TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

OPTIMISTIC CONCURRENCY CONTROL TECHNIQUES

In optimistic concurrency control techniques, also known as validation or certification techniques, no *checking* is done while the transaction is executing. Several proposed concurrency control methods use the validation technique. In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction. At the end of transaction execution, a validation phase checks whether any of the transaction"s updates violate serializability.

Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase**: A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase:** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase:** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.