

## UNIT - III

**Transaction Management:** Definition, properties of transaction, types of transactions, distributed concurrency control: serializability, concurrency control mechanisms & algorithms, time - stamped & optimistic concurrency control Algorithms, deadlock Management.

---

### Transaction Management: Definition

A **transaction** is a program including a collection of database operations, executed as a logical unit of data processing. The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data. It is an atomic process that is either performed into completion entirely or is not performed at all. A transaction involving only data retrieval without any data update is called read-only transaction.

Each high level operation can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks –

- **read\_item()** – reads data item from storage to main memory.
- **modify\_item()** – change value of item in the main memory.
- **write\_item()** – write the modified value from main memory to storage.

Database access is restricted to read\_item() and write\_item() operations. Likewise, for all transactions, read and write forms the basic database operations.

### Properties of transaction:

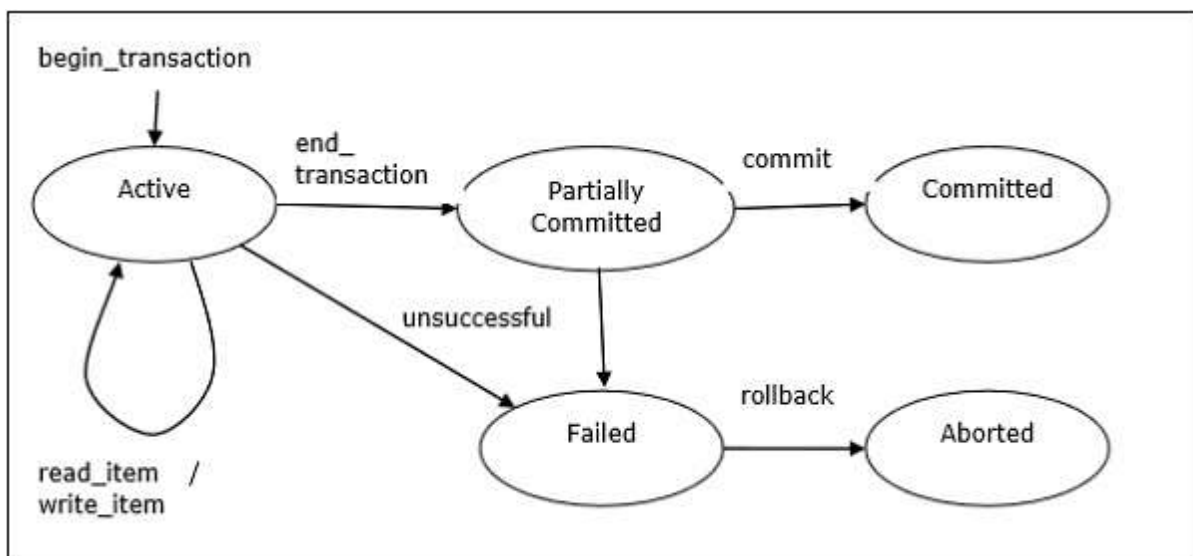
Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** – This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.
- **Consistency** – A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database.
- **Isolation** – A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.
- **Durability** – If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

### Types of transactions:

A transaction may go through a subset of five states, active, partially committed, committed, failed and aborted.

- **Active** – The initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations.
- **Partially Committed** – The transaction enters this state after the last statement of the transaction has been executed.
- **Committed** – The transaction enters this state after successful completion of the transaction and system checks have issued commit signal.
- **Failed** – The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail.
- **Aborted** – This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began.



### **Distributed concurrency control: serializability:**

In **database** systems and transaction processing (transaction **management**) **distributed concurrency control** refers primarily to the **concurrency control** of a **distributed database**. A major goal for **distributed concurrency control** is **distributed serializability** (or global **serializability** for multidatabase systems).

**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.

A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule.

Equivalence of Schedules

Equivalence of two schedules can be of the following types –

- **Result equivalence** – Two schedules producing identical results are said to be result equivalent.
- **View equivalence** – Two schedules that perform similar action in a similar manner are said to be view equivalent.
- **Conflict equivalence** – Two schedules are said to be conflict equivalent if both contain the same set of transactions and has the same order of conflicting pairs of operations.

### **Concurrency control mechanisms & algorithms:**

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

#### **Locking Based Concurrency Control Protocols**

Locking-based concurrency control protocols use the concept of locking data items. A **lock** is a variable associated with a data item that determines whether read/write operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

#### **One-phase Locking Protocol**

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

#### **Two-phase Locking Protocol**

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the expanding or the **growing phase**. In the second phase, the transaction releases the locks and cannot request any new locks. This is called the **shrinking phase**.

Every transaction that follows two-phase locking protocol is guaranteed to be serializable. However, this approach provides low parallelism between two conflicting transactions.

#### **Timestamp Concurrency Control Algorithms**

Timestamp-based concurrency control algorithms use a transaction's timestamp to coordinate concurrent access to a data item to ensure serializability. A timestamp is a unique identifier given by DBMS to a transaction that represents the transaction's start time.

These algorithms ensure that transactions commit in the order dictated by their timestamps. An older transaction should commit before a younger transaction, since the older transaction enters the system before the younger one.

Timestamp-based concurrency control techniques generate serializable schedules such that the equivalent serial schedule is arranged in order of the age of the participating transactions.

Some of timestamp based concurrency control algorithms are –

- Basic timestamp ordering algorithm.
- Conservative timestamp ordering algorithm.
- Multiversion algorithm based upon timestamp ordering.

Timestamp based ordering follow three rules to enforce serializability –

- **Access Rule** – When two transactions try to access the same data item simultaneously, for conflicting operations, priority is given to the older transaction. This causes the younger transaction to wait for the older transaction to commit first.
- **Late Transaction Rule** – If a younger transaction has written a data item, then an older transaction is not allowed to read or write that data item. This rule prevents the older transaction from committing after the younger transaction has already committed.
- **Younger Transaction Rule** – A younger transaction can read or write a data item that has already been written by an older transaction.

### **Optimistic Concurrency Control Algorithm**

In systems with low conflict rates, the task of validating every transaction for serializability may lower performance. In these cases, the test for serializability is postponed to just before commit. Since the conflict rate is low, the probability of aborting transactions which are not serializable is also low. This approach is called optimistic concurrency control technique.

In this approach, a transaction's life cycle is divided into the following three phases –

- **Execution Phase** – A transaction fetches data items to memory and performs operations upon them.
- **Validation Phase** – A transaction performs checks to ensure that committing its changes to the database passes serializability test.
- **Commit Phase** – A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase –

**Rule 1** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is reading the data item which  $T_j$  is writing, then  $T_i$ 's execution phase cannot overlap with  $T_j$ 's commit phase.  $T_j$  can commit only after  $T_i$  has finished execution.

**Rule 2** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is writing the data item that  $T_j$  is reading, then  $T_i$ 's commit phase cannot overlap with  $T_j$ 's execution phase.  $T_j$  can start executing only after  $T_i$  has already committed.

**Rule 3** – Given two transactions  $T_i$  and  $T_j$ , if  $T_i$  is writing the data item which  $T_j$  is also writing, then  $T_i$ 's commit phase cannot overlap with  $T_j$ 's commit phase.  $T_j$  can start to commit only after  $T_i$  has already committed.

## **Concurrency Control in Distributed Systems**

In this section, we will see how the above techniques are implemented in a distributed database system.

### **Distributed Two-phase Locking Algorithm**

The basic principle of distributed two-phase locking is same as the basic two-phase locking protocol. However, in a distributed system there are sites designated as lock managers. A lock manager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.

Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types –

- **Centralized two-phase locking** – In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.
- **Primary copy two-phase locking** – In this approach, a number of sites are designated as lock control centers. Each of these sites has the responsibility of managing a defined set of locks. All the sites know which lock control center is responsible for managing lock of which data table/fragment item.
- **Distributed two-phase locking** – In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its local site. The location of the lock manager is based upon data distribution and replication.

### **Time - stamped & optimistic concurrency control Algorithms:**

#### **Distributed Timestamp Concurrency Control**

In a centralized system, timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site's local physical/logical clock readings cannot

be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site's clock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager. During transaction, a transaction manager sends a lock request to the site's scheduler. The scheduler puts the request to the corresponding queue in increasing timestamp order. Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first.

### **Conflict Graphs**

Another method is to create conflict graphs. For this transaction classes are defined. A transaction class contains two set of data items called read set and write set. A transaction belongs to a particular class if the transaction's read set is a subset of the class' read set and the transaction's write set is a subset of the class' write set. In the read phase, each transaction issues its read requests for the data items in its read set. In the write phase, each transaction issues its write requests.

A conflict graph is created for the classes to which active transactions belong. This contains a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and denotes conflicts within the class. A horizontal edge connects two nodes across two classes and denotes a write-write conflict among different classes. A diagonal edge connects two nodes across two classes and denotes a write-read or a read-write conflict among two classes.

The conflict graphs are analyzed to ascertain whether two transactions within the same class or across two different classes can be run in parallel.

### **Distributed Optimistic Concurrency Control Algorithm**

Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm. For this extension, two rules are applied –

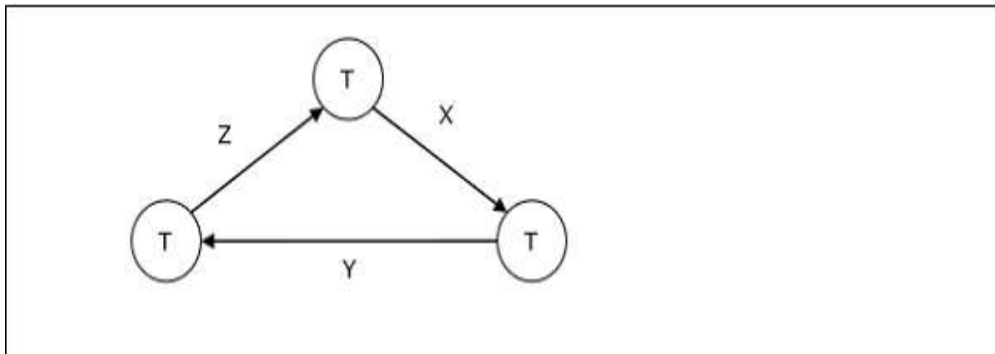
**Rule 1** – According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

**Rule 2** – According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

### **Deadlock Management:**

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. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



## Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely –

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

All of the three approaches can be incorporated in both a centralized and a distributed database system.

### 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.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any 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.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

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.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-for graph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are –

- Choose the youngest transaction.
- Choose the transaction with fewest data items.
- Choose the transaction that has performed least number of updates.
- Choose the transaction having least restart overhead.
- Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

### Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.



## **Transaction Location**

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

## **Transaction Control**

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like –

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

## **Distributed Deadlock Prevention**

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks –

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

## **Distributed Deadlock Avoidance**

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur –

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows –

- **Distributed Wound-Die**
  - If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
  - If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.
- **Distributed Wait-Wait**
  - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
  - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

## **Distributed Deadlock Detection**

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector** – One site is designated as the central deadlock detector.
- **Hierarchical Deadlock Detector** – A number of deadlock detectors are arranged in hierarchy.
- **Distributed Deadlock Detector** – All the sites participate in detecting deadlocks and removing them.