Module No.6

Transactions Management and Concurrency and Recovery

Transaction concept, Transaction states, ACID properties, Transaction Control Commands, Concurrent Executions, Serializability-Conflict and View, Concurrency Control: Lock-based, Timestamp-based protocols, Recovery System: Log based recovery, Deadlock handling

Transaction

A sequence of many actions is considered one atomic unit of work. A transaction is a collection of operations involving data items in a database.

It is the bundle of all the instructions of a logical operation. A transaction usually means that the data in the database has changed. One of the major uses of DBMS is to protect the user's data from system failures. It is done by ensuring that all the data is restored to a consistent state when the computer is restarted after a crash. The transaction is any one execution of the user program in a DBMS. One of the important properties of the transaction is that it contains a finite number of steps. Executing the same program multiple times will generate multiple transactions.

Example: Consider the following example of transaction operations to be performed to withdraw cash from an ATM vestibule.

Steps for ATM Transaction

- Transaction Start.
- Insert your ATM card.
- Select a language for your transaction.
- Select the Savings Account option.
- Enter the amount you want to withdraw.
- Enter your secret pin.
- Wait for some time for processing.
- Collect your Cash.
- Transaction Completed.

A transaction can include the following basic database access operation.

- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W):** Write the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

  Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from Hard Disk.

- R(A) -- 500 // Accessed from RAM.
- A = A-50 // Deducting 50₹ from A.
- W(A)--450 // Updated in RAM.
- R(B) -- 800 // Accessed from RAM.
- B=B+50 // 50₹ is added to B's Account.
- W(B) --850 // Updated in RAM.
- commit // The data in RAM is taken back to Hard Disk.

All instructions before committing come under a partially committed state and are stored in RAM. When the commit is read the data is fully accepted and is stored on Hard Disk.

If the data is failed anywhere before committing we have to go back and start from the beginning. We can't continue from the same state. This is known as Roll Back.

A DBMS must ensure four important transaction properties to maintain data in the face of concurrent access and system failures, that is ACID properties.

ACID properties

A = Atomicity
C = Consistency
I = Isolation
D = Durability

Atomicity

Atomicity requires that each transaction is all or nothing. If one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

Consistency

If each transaction is consistent and the database starts as consistent, it ends up as consistent.

Isolation

The execution of one transaction is isolated from that of another transaction. It ensures that concurrent transaction execution results in a system state that would be obtained if the transaction were executed serially, i.e., one after the other.

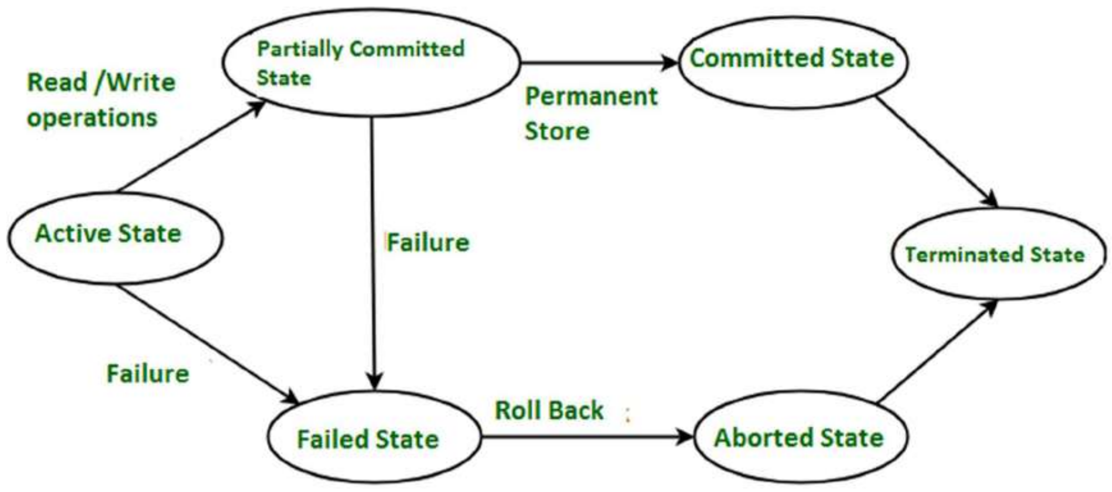
Durability

Durability means that once a transaction has been committed, it will remain even in the event of power loss, crashes, or errors.

For instance, once a group of SQL statements executes in a relational database, the results need to be stored permanently.

Transaction States

The states through which a transaction passes during its lifetime. These are the states that describe the current state of the transaction and how we will proceed with the processing. These states govern the rules determining whether the transaction will commit or abort.



Transaction States in DBMS

Active state

A transaction will be called in an active state if its instructions are executed. This is because all the changes made by this transaction are now stored in the buffer in the main memory.

Partially committed state

After the last instruction of a transaction has been executed, it enters this partially committed state. After entering the state, the transaction is said to be partially committed because it has not performed the commit operation. However, the transaction is not considered fully committed because all the transaction changes are still stored in the buffer in the main memory.

Committed state

After all the changes executed by the transaction have been committed and stored successfully in the database, it enters into a state called the committed state. Now, the transaction is considered fully committed.

Abort state (Rollback state)

When a transaction is being executed in the active state or partially committed state, and some failure occurs, due to which it becomes impossible to continue the execution, it enters into a failed state.

Failed State

After the transaction is failed and enters into a failed state, all the changes made by this transaction have to be undone. To undo the changes made by this transaction, it becomes necessary to roll back all the transaction operations. After the transaction fully rolls back, it enters into an aborted state.

Terminated state

When a transaction successfully performs the redo operation in case of a committed transaction or performs the undo operation in case of a failed transaction, it is said to be in the terminated state. Therefore, to maintain the correctness of the databases, it will execute all the transactions successfully, or none of the transactions will be executed in case any failure occurs.

Concurrency Control

The process of managing the simultaneous execution of transactions in a shared database is known as **concurrency control**. Concurrency control ensures that correct results for concurrent operations are generated while getting those results as quickly as possible.

Need for Concurrency Control

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.

Lost Update

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Dirty Read Problems

This problem occurs when one transaction changes the value while the other reads the value before committing or rolling back by the first transaction.

Inconsistent Retrievals

This problem occurs when a transaction accesses data before and after another transaction(s) finishes working with such data.

We need concurrence control, when

- The amount of data is sufficiently great that at any time, only a fraction of the data can be in primary memory, and the rest should be swapped from secondary memory as needed.
- Even if the entire database can be present in primary memory, there may be multiple processes.

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

- Lock-Based Protocols
- Two Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols

1. **Lock-based Protocols**

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.
A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Concurrency Control with Locking

If concurrency control with the locking technique is used, then locks prevent multiple transactions from accessing the items concurrently.

- Access on data only if TA 'has lock' on data.
- Transactions request and release locks.
- The schedule allows or differs operations based on the lock table.

Lock: A lock is a variable associated with a data item that describes the item's status concerning possible operations that can be applied to it. (e.g., read lock, write lock). Locks are used to synchronizing access by concurrent transactions to the database items. There is one lock per database item.

Purpose of Concurrency Control with Locking

- To enforce isolation among conflicting transactions.
- To preserve database consistency.
- To resolve read-write, write-read, and write-write conflicts.

Locking is an operation that secures permission to read and permission to write a data item.

Binary Locks: A Binary lock on a data item can either locked or unlocked states. If a database item X is locked, then X cannot be accessed by any other database operation.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock. There are three states:- Read locked / Writelocked / Unlocked. Several transactions can access the same item X for reading (shared lock); however, if any transactions want to write item X, the transaction must acquire an exclusive lock on the item.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

Problems Arising with Locks: There are two problems that are possible when using locks to control the concurrency among transactions.

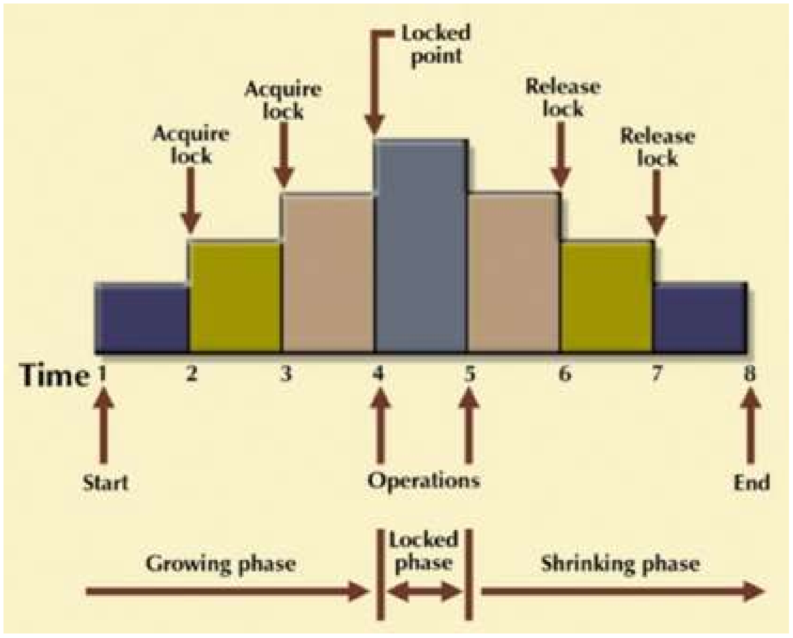
- **Deadlock:** Two or more competing transactions are waiting for each other to complete to obtain a missing lock.
- **Starvation:** A transaction is continually denying access to a given data item.

2. Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.
This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.

- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

3. Timestamp-based Protocols

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.
The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are there transactions T1, T2, and T3.
T1 has entered the system at time 0010
T2 has entered the system at 0020
T3 has entered the system at 0030
Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. An 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 ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering –
Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that $\mathbf{TS(T_i) < TS(T_j)}$. The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in the following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
    - If ***R_TS(X) > TS(T)*** or if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).

2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:
   - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
   - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

Starvation is possible if the same transaction is restarted and continually aborted

4. Validation Based Protocol

Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.
The Validation based Protocol is performed in the following three phases:

1. Read Phase
2. Validation Phase
3. Write Phase

Read Phase

In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.

Validation Phase

In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.

Write Phase

In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.

Log-Based Recovery

- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.
    1. <Tn, Start>
- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.
    1. <Tn, City, 'Noida', 'Bangalore' >
- When the transaction is finished, then it writes another log to indicate the end of the transaction.
    1. <Tn, Commit>

There are two approaches to modify the database:

1. Deferred database modification:

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

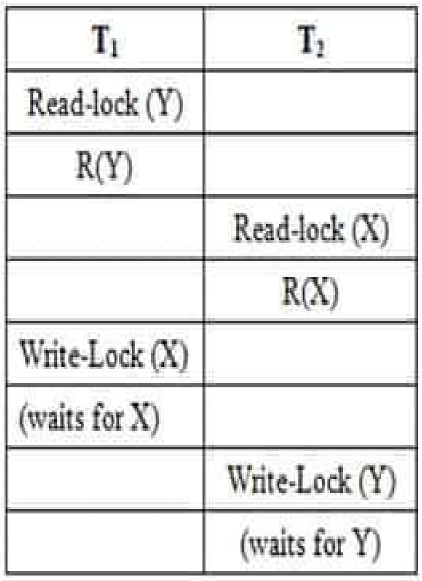
When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.

2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

What is Deadlock?

A deadlock is a condition in which two or more transactions are waiting for each other deadlock (T_1 and T_2). An example is given below.



T_1	T_2
Read-lock (Y)	
R(Y)	
	Read-lock (X)
	R(X)
Write-Lock (X)	
(waits for X)	
	Write-Lock (Y)
	(waits for Y)

Deadlock Prevention: A transaction locks all data items; it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. Conservative two-phase locking uses this approach.

Deadlock Detection and Resolution: In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycles. If a cycle exists, one transaction involved is selected (victim) and rolled back.

- A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph.
- When a chain like T_i waits for T_j, T_j waits for T_K, and T_k waits for T_i or $T_{j, \text{this}}$ creates a cycle. One of the transactions of the cycle is selected and rolled back.

Deadlock Avoidance: There are many variations of the two-phase locking algorithm. Some avoid deadlock by not letting the cycle to complete. This is because as soon as the algorithm discovers that blocking a translation is likely to create a cycle, it rolls back the transaction.

The following schemes use transaction timestamps for the sake of deadlock avoidance.

Wait-die scheme (Non-preemptive): Older transactions may wait for the younger ones to release data items. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring the needed data item.

Wound-wait scheme (Preemptive): Older transaction bounds (forces rollback) of the younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than the wait-die scheme.

Starvation: Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution scheme, it is possible that the same transaction may consistently be selected as a victim and rolled back.

Time-stamp-Based Concurrency Control Algorithm: This is a different approach that guarantees serializability and involves using transaction time-stamps to order transaction execution for an equivalent serial schedule.

Time-stamp: A time-stamp is a unique identifier created by the DBMS to identify a transaction.

- Time-stamp is a monotonically increasing variable (integer) indicating the age of an operation or a transaction.
- A larger time-stamp value indicates a more recent event or operation.

Starvation *versus* Deadlock

Starvation	**Deadlock**
Starvation happens if the same transaction is always chosen as the victim.	A deadlock is a condition in which two or more transactions are waiting for each other.
It occurs if the waiting scheme for locked items is unfair, prioritizing some transactions over others.	A situation where two or more transactions cannot proceed because each is waiting for one of the others to do something.
Starvation is also known as a lived lock.	Deadlock is also known as circular waiting.
Switch priorities so that every threat has a chance to have high priority.	Use FIFO order among competing requests.
It is a situation where transactions are waiting for each other.	Acquire locks are in a predefined order.

It means that the transaction goes in a state where the transaction never progresses.	Acquire locks at one before starting

The algorithm associates each database item X with two Time Stamp (TS) values

Read_TS(X): The read timestamp of item X; this is the largest time-stamp among all the time-stamps of transactions that have successfully read item X.

Write_TS (X): The write time-stamp of item X; this is the largest time-stamp among all the time-stamps of transactions that have successfully written item X.

There are two Time-Stamp-based Concurrency Control Algorithms.

Basic Time-stamp Ordering (TO): Whenever some transaction T tries to issue an R(X) or a W(X) operation, the basic time-stamp ordering algorithm compares the time-stamp of T with read_TS (X) and write_TS (X) to ensure that the time-stamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted. 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 T_2 that may have used a value written by T_1 must also be rolled back. This effect is known as cascading rollback. The concurrency control algorithm must check whether conflicting operations violate the time-stamp ordering in the following two cases.

Transaction T issues a W(X) operation: If read_TS (X) > TS (T) or if write_TS (X) > TS (T), then a younger transaction has already read the data item, so abort and roll back T and reject the operation.

If the condition in the above part does not exist, then execute W(X) of T and set write_TS (X) to TS (T).

Transaction T issues a R (X) operation: If write_TS (X) > TS (T), then a younger transaction has already been written to the data item, so abort and roll back T and reject the operation.

If write_TS

Strict Time-stamp Ordering (TO): A variation of basic time-stamp ordering, called strict time-stamp ordering, ensures that the schedules are both strict (for easy recoverability) and serializable (conflict).

Transaction T issues a W(X) operation: If TS (T) > read_TS (X), then delay T until the transaction

Transaction T issues a R(X) operation: If TS (T) > write_TS (X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Thomas's Write Rule:

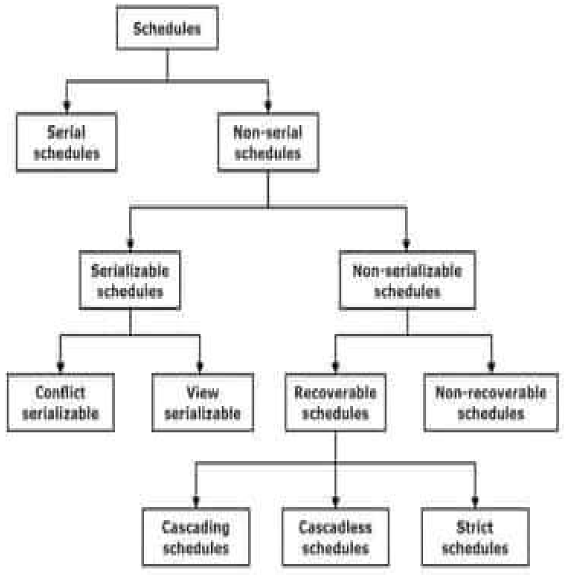
- If read_TS (X) > TS (T), then abort and roll back T and reject the operation.
- If write_TS (X) > TS (T), then just ignore the write operation and continue execution.
- This is because the most recent writes count in the case of two consecutive writes. If the conditions given in (i) and (ii) above do not occur, then execute W(X) of T and set write_TS (X) to TS(T).

What is the Schedule in Transaction and Concurrency Control Protocol in DBMS?

A schedule (or history) is a model to describe the execution of transactions running in the system. When multiple transactions are executed concurrently in an interleaved fashion, then the order of execution of operations from the various transactions is known as a schedule, or we can say that a schedule is a sequence of reading, writing, aborting, and committing operations from a set of transactions.

Classification of Schedules Based on Serializability

A schedule in which the different transactions are not interleaved (i.e., transactions are executed from start to finish one by one). Serial schedules and concurrent schedules are further classified as shown in the classification diagram:



Serial Schedule

A schedule in which a new transaction is allowed to begin only after completing the previous transaction. No middle interleaving is allowed.

Serial schedule	
T_1	T_2
	R(A)
	W(A)
R(B)	
W(B)	

Serial Schedule	
T_1	T_2
R(B)	
W(B)	
	R(A)
	W(A)

Complete Schedule: A schedule that contains either a commit or an abort action for each transaction. Consequently, a complete schedule will not contain any active transactions at the end of the schedule.

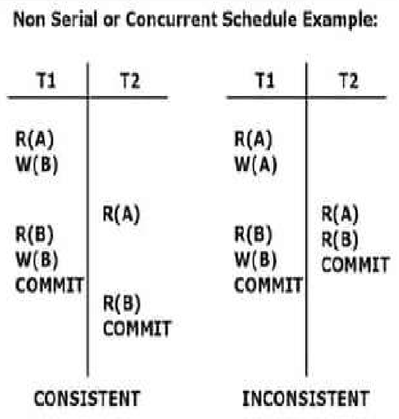
Complete schedule	
T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

Complete Schedule	
T_1	T_2
R(A)	
W(A)	
	R(B)
Commit	
	W(B)
	Abort

Complete Schedule	
T_1	T_2
R(A)	
W(A)	
Commit	
	R(B)
	W(B)
	Abort

Non-serial Schedules: Non-serial schedules are interleaved schedules that improve the system's performance (i.e., throughput and response time). But concurrency or interleaving operations in schedules might lead the database to an inconsistent state. We need to seek to identify schedules that are:

1. As fast as interleaved schedules.
2. As consistent as serial schedules.


Non Serial or Concurrent Schedule Example:

T1	T2
R(A)	
W(B)	
	R(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	COMMIT

CONSISTENT

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	R(B)
W(B)	COMMIT
COMMIT	

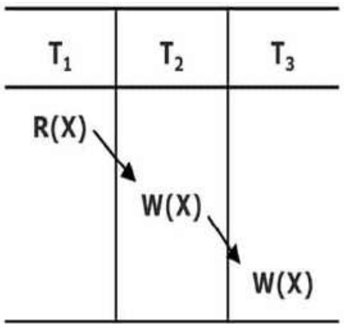
INCONSISTENT

Conflicting Operations

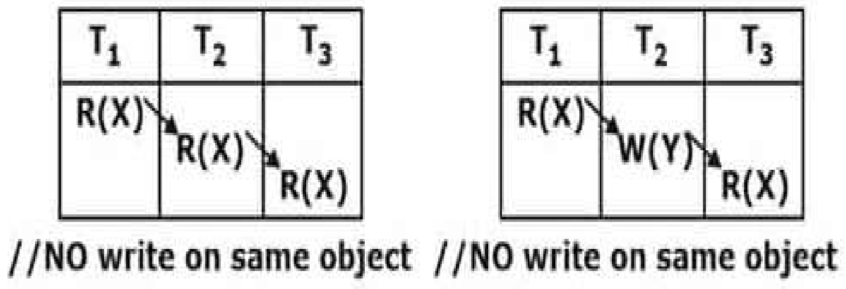
When two or more transactions in a non-serial schedule execute concurrently, some conflicting operations may occur. Two operations are said to be conflicting if they satisfy the following conditions.

- The operations belong to different transactions.
- At least one of the operations is a write operation.
- The operations access the same object or item.

The following set of operations is conflicting.



While the following sets of operations are not conflicting



Serializable Schedule

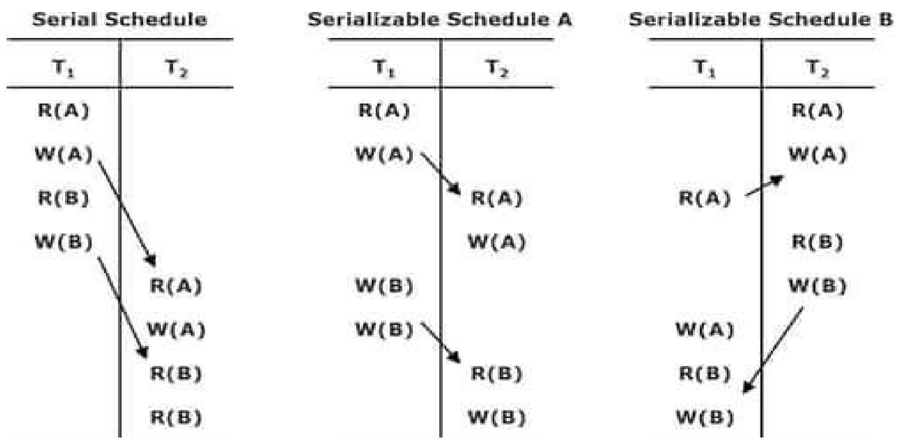
A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

A non-serial schedule S is serializable is equivalent to saying that it is correct because it is equivalent to a serial schedule. There are two types of serializable schedules.

1. Conflict serializable schedule
2. View serializable schedule

Conflict Serializable Schedule

When the schedule (S) is conflict equivalent to some serial schedule (Sï), that schedule is called a conflict serializable schedule. In such a case, we can reorder the non-conflicting operations in S until we form the equivalent serial schedule S'.

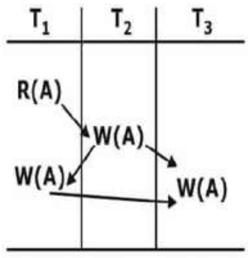


Conflict Equivalence: The schedules S_1 and S_2 are said to be conflict equivalent if the following conditions are satisfied:

- Both schedules S_1 and S_2 involve the same set of transactions (including the ordering of operations within each transaction).
- The order of each pair of conflicting actions in S_1 and S_2 is the same.

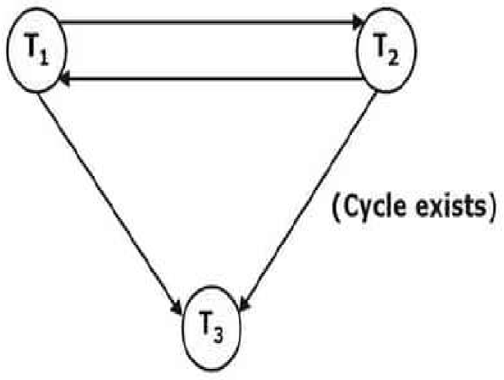
Testing for Conflict Serializability of a Schedule: A simple algorithm can be used to test a schedule for conflict serializability. This algorithm constructs a precedence graph (or serialization graph), a directed graph. A precedence graph for a schedule S contains

1. A node for each committed transaction in S.
2. An edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j's operations.



Conflict serializability

A schedule S is a conflict serializable if and only if its precedence graphs are acyclic.

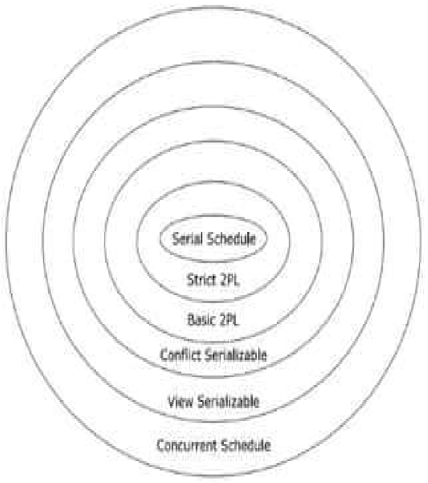


Precedence graph

Note: The above example of schedule does not conflict serializable schedule.

View Serializable Schedule

A schedule is viewed as serializable if it is equivalent to some serial schedule. Conflict serializable ï View serializable, but not vice-versa.



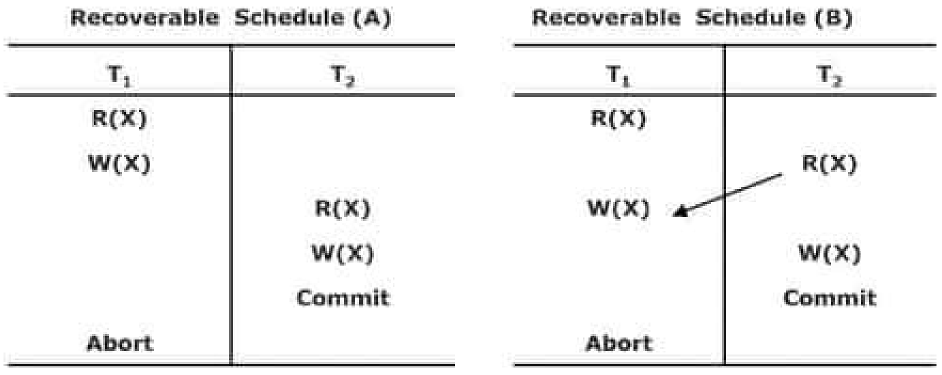
View Equivalence: Two schedules S_1 and $S_{2,}$ are view equivalent,

1. T_i reads the initial value of database object A in schedule S_1 then T_i also reads the initial value of database object A in schedule S_2.
2. T_i reads the value of A written by T_j in schedule S_1 then T_i also reads the value of A written by T_j in schedule S_2.
3. T_i writes the final value of A in schedule S_l then T_i also writes the final value of A in S_2.

In the above example, both schedules S_1 and S_2 are view equivalent. So, they view serializable schedules. But the S_1 schedule does not conflict serializable schedule, and S_2 is not conflicted serializable schedule because the cycle is not formed.

Recoverable Schedule

Once a transaction T is committed, rolling back T should never be necessary. The schedules under this criteria are called recoverable schedules, and those that do not are called non-recoverable. A schedule S is recoverable if no transaction T in S commits until all transactions T that have written an item T have committed.



Recoverable Schedule (A)

T₁	T₂
R(X)	
W(X)	
	R(X)
	W(X)
	Commit
Abort	

(Commit after parent of dirty read)
(Recoverable)

Recoverable Schedule (B)

T₁	T₂
R(X)	
	R(X)
W(X)	
	W(X)
	Commit
Abort	

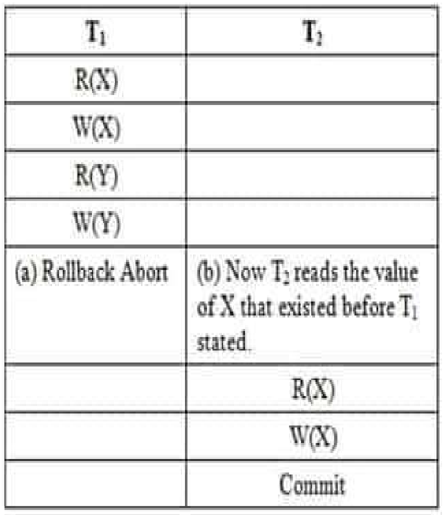
Remove dirty read (Recoverable) not serializable schedule (Order of conflicting actions has changed)

Cascade-less Schedule

These schedules avoid cascading rollbacks. Even if a schedule is recoverable to recover correctly from the failure of transaction T_i, we may have to roll back several transactions. This phenomenon in which a single transaction failure leads to a series of transaction rollbacks is called **cascading rollbacks.**

Cascading rollback is undesirable since it leads to undoing a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascade-less **schedules**.

Cascade-less Schedule

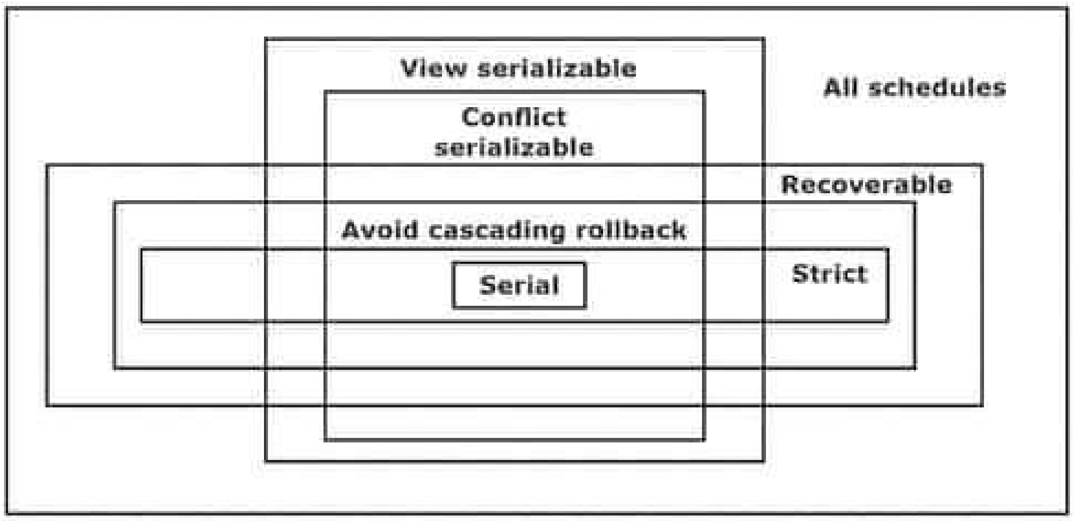


T_1	T_2
R(X)	
W(X)	
R(Y)	
W(Y)	
(a) Rollback Abort	(b) Now T_2 reads the value of X that existed before T_1 stated.
	R(X)
	W(X)
	Commit

Strict Schedule

A schedule is strict,

- If overriding uncommitted data is not allowed.
- Formally, if it satisfies the following conditions
    1. T_j reads a data item X after T_i has terminated (aborted or committed).
    2. T_j writes a data item X after T_i has terminated (aborted or committed).



Characterizing Schedules through Venn Diagram