Serializability and Concurrency Control in DBMS

Dr. Anish Soni

Serializability in DBMS

- In DBMS, serializability ensures the database remains correct even when multiple transactions run at the same time.
- It makes transactions behave as if they were executed one after another in some order.
- This prevents conflicts, maintains data consistency, and ensures reliable results, just like in a single, orderly sequence of transactions.

Schedules in DBMS

- To understand the idea of serializability, we must first understand what a schedule is.
- In DBMS, a schedule is a sequence of transaction operations (like reads or writes). These operations are arranged in a specific order.
- It is determines how transactions are executed. Specially when multiple transactions run concurrently.

Serial Schedule

- When we are talking one schedules we must consider serial and non-serial schedules. A serial schedule processes transactions one after another without overlapping. For example: Transaction T1 runs entirely before Transaction T2 begins.
- This kind of schedule is straightforward and always consistent because transactions are isolated from one another. But, it can be slow due to the lack of concurrency. Here is an **example of a serial schedule** –

Serial Schedule

Schedule 1: T1	Operation	Schedule 2: T2	Operation
		T2: Read(B)	Read B
		T2: Write(B)	Write B
		T2: Read(A)	Read A
		T2: Write(A)	Write A
T1: Read(A)	Read A		
T1: Write(A)	Write A		
T1: Read(B)	Read B		
T1: Write(B)	Write B		

In this example, T2 completes both its operations before T1 starts. Since transactions do not overlap, the schedule is guaranteed to be consistent.

Non-Serial Schedule

- A non-serial schedule allows transactions to run concurrently and may access the same data.
- To ensure database consistency, it must be serializable, meaning it should produce the same result as some serial (one-by-one) execution.

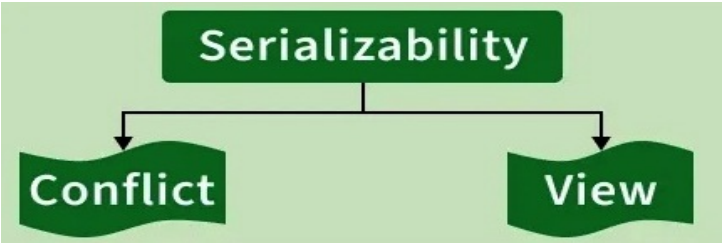
Non-Serial Schedule

Schedule 1: T1	Operation	Schedule 2: T2	Operation
T1: Read(A)	Read A		
T1: Write(A)	Write A		
		T2: Read(B)	Read B
		T2: Write(B)	Write B
T1: Read(B)	Read B		
T1: Write(B)	Write B		
		T2: Read(A)	Read A
		T2: Write(A)	Write A

Here, transactions T1 and T2 run concurrently. While this improves performance. There is no immediate guarantee of consistency.

Types of Serializability

- There are two ways to check whether any non-serial schedule is serializable.



Conflict Serializability ensures that a concurrent schedule produces the same result as some serial execution **by reordering non-conflicting operations.** It maintains data consistency and is stricter than View Serializability, which allows more flexibility but still preserves correctness.

Non-conflicting operations: Two operations are considered non-conflicting if they operate on separate data items, or if they involve the same data item but both are read operations.

Conflicting Operations

Two operations are said to be conflicting if all conditions are satisfied:

- They belong to different transactions
- They operate on the same data item
- At least one of them is a write operation

Operation by T_i	Operation by T_j	Conflict Status
read(Q)	read(Q)	No Conflict
read(Q)	write(Q)	Conflict
write(Q)	read(Q)	Conflict
write(Q)	write(Q)	Conflict

Conflicting Operations

Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.

Example 2: Similarly Operations W(X) of T1 and W(X) of T2 are conflicting operations.

Example 3: Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.

Example 4: Similarly R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

Example 5: Similarly W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

Conflict Serializable check

Lets check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called Conflict Serializable schedule. Lets take few examples of schedules.

Example of Conflict Serializability

Lets consider this schedule:

```
T1          T2
-----       ------
R(A)
R(B)
            R(A)
            R(B)
            W(B)
W(A)
```

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Lets take another example:

```
T1              T2
------          -------
R(A)
                R(A)
                R(B)
                W(B)
R(B)
W(A)
```

Lets **swap non-conflicting operations**:

After swapping R(A) of T1 and R(A) of T2 we get:

```
T1              T2
------          -------
                R(A)
R(A)
                R(B)
                W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and R(B) of T2 we get:

```
T1              T2
------          -------
                R(A)
                R(B)
R(A)
                W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and W(B) of T2 we get:

```
T1              T2
------          -------
                R(A)
                R(B)
                W(B)
R(A)
R(B)
W(A)
```

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.

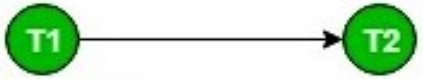
- Consider the schedule S:
- S: r1(x) r1(y) w2(x) w1(x) r2(y)

Creating Precedence Graph

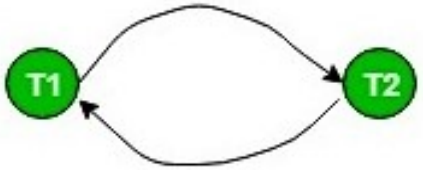
Step 1: Make two nodes corresponding to Transaction T_1 and T_2.



Step 2: For the conflicting pair r1(x) w2(x), where r1(x) happens before w2(x), draw an edge from T_1 to T_2.



Step 3: For the conflicting pair w2(x) w1(x), where w2(x) happens before w1(x), draw an edge from T_2 to T_1.



Since the graph is cyclic, we can conclude that it is not conflict serializable to any schedule serial schedule. Let us try to infer a serial schedule from this graph using topological ordering.

View Serializability in DBMS

- In database systems, concurrent execution of transactions is used to improve resource utilization and system throughput. However, concurrency can lead to inconsistencies in the database if not handled properly.
- **View Serializability** ensures that even though transactions run concurrently, their outcome will be identical to the result achieved if the transactions were executed one by one in a specific order.
- It is a concept in concurrency control that checks whether a non-serial schedule produces the same final output as a serial schedule.
- It ensures data consistency by verifying that the initial reads, updated reads, and final writes of transactions are the same as in a serial schedule.

View Serializability in DBMS

- A schedule is view serializable if it is **view equivalent** to some serial schedule, meaning it produces the exact same final result even if the intermediate step orders differ.
- **View Equivalence Conditions:** Two schedules are view equivalent if they satisfy three rules:

1. **Initial Read:** Each transaction must read the initial value of a data item from the same transaction in both schedules.
2. **Updated Read (Read-Write Sequence):**

- If transaction T_j reads a data item written by T_i

in one schedule, it must do the same in the other schedule.

- **Final Write:** The last write operation on any data item must be performed by the same transaction in both schedules.

Let us consider the following transaction schedule and test it for View Serializability.

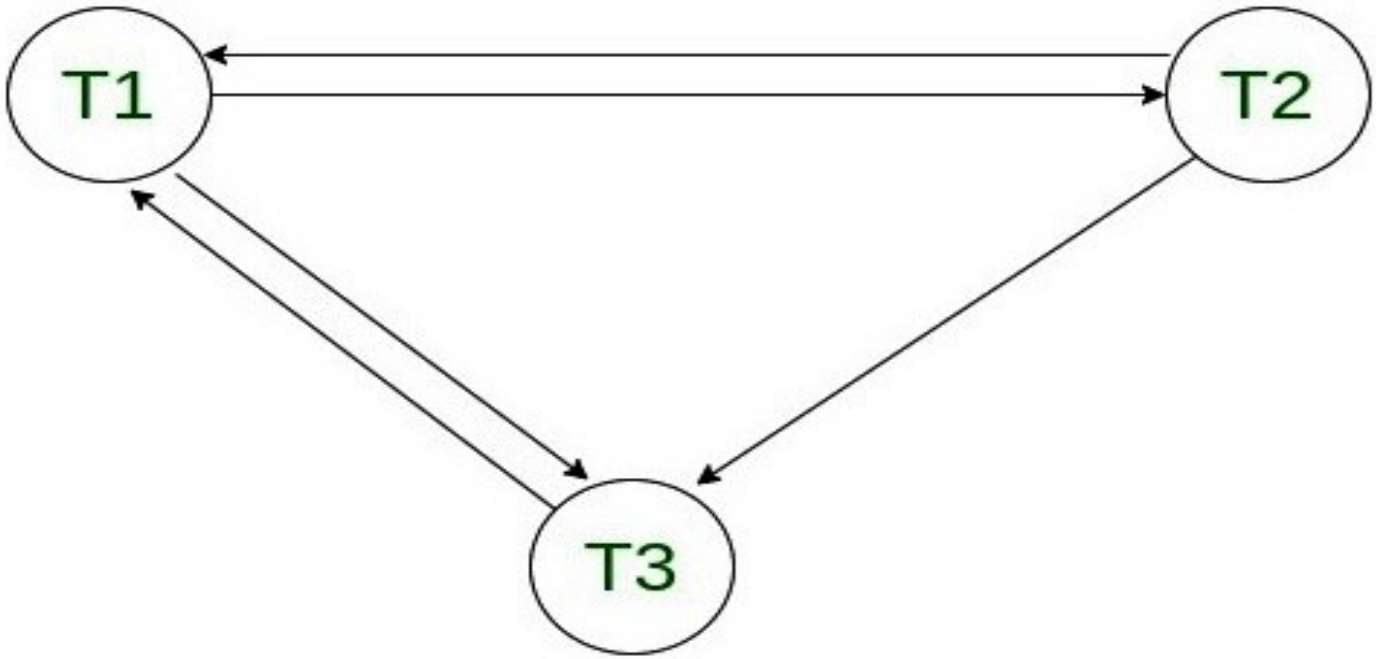
T1	T2	T3
R(A)		
	W(A)	
		R(A)
W(A)		
		W(A)

As we know that if a schedule is Conflict Serializable, then it is View Serializable also. So first let us check for Conflict Serializability.

The conflicting operations for this schedule are -

1. R1(A) and W2(A) [T1 -> T2]
2. R1(A) and W3(A) [T1 -> T3]
3. W2(A) and R3(A) [T2 -> T3]
4. W2(A) and W1(A) [T2 -> T1]
5. W2(A) and W3(A) [T2 -> T3]
6. R3(A) and W1(A) [T3 -> T1]
7. W3(A) and W1(A) [T1 -> T3]

Constructing the precedence graph for conflicting operations in the schedule.

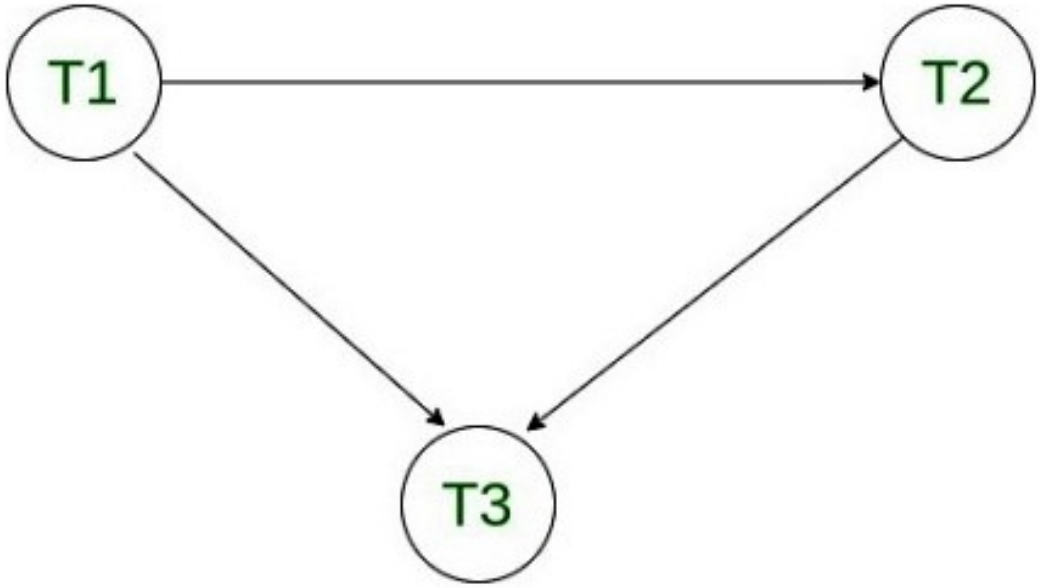


As we can see that there is a cycle in the precedence graph, it means that the given schedule is not Conflict Serializable. Now, on checking for blind write we get that there exists a blind write W2(A) in the given schedule. Thus, the schedule may or may not be View Serializable.

In order to check for View Serializability, we will draw a Dependency Graph of the schedule. From the given schedule we gather the following points :

1. T1 reads A before T2 updates A thus, T1 must execute before T2.
2. T3 does the final update on A thus, it must execute in the end.

Constructing the dependency graph.



As there exists no cycle in the graph, we can say that the given schedule is View Serializable.
The serializable schedule is T1 -> T2 -> T3.

Conflict Serializability	View Serializability
Two schedules are said to be conflict equivalent if all the conflicting operations in both the schedule get executed in the same order. If a schedule is a conflict equivalent to its serial schedule then it is called Conflict Serializable Schedule.	Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules. If a schedule is view equivalent to its serial schedule then it is called View Serializable Schedule.
If a schedule is view serializable then it may or may not be conflict serializable.	If a schedule is conflict serializable then it is also view serializable schedule.
Conflict equivalence can be easily achieved by reordering the operations of two transactions therefore, Conflict Serializability is easy to achieve.	View equivalence is rather difficult to achieve as both transactions should perform similar actions in a similar manner. Thus, View Serializability is difficult to achieve.
For a transaction T1 writing a value A that no one else reads but later some other transactions say T2 write its own value of A, W(A) cannot be placed under positions where it is never read.	If a transaction T1 writes a value A that no other transaction reads (because later some other transactions say T2 writes its own value of A) W(A) can be placed in positions of the schedule where it is never read.

Introduction- Concurrency Control

- In a Database Management System (DBMS), Concurrency control is a mechanism in DBMS that allows simultaneous execution of transactions while maintaining ACID properties - Atomicity, Consistency, Isolation and Durability.
- It maintains the integrity, accuracy and reliability of data when multiple users or processes perform read/write operations concurrently.
- It helps manage issues like:

➢ Conflicting operations on shared data.

➢ Inconsistent database states.

➢ Lost or incorrect updates

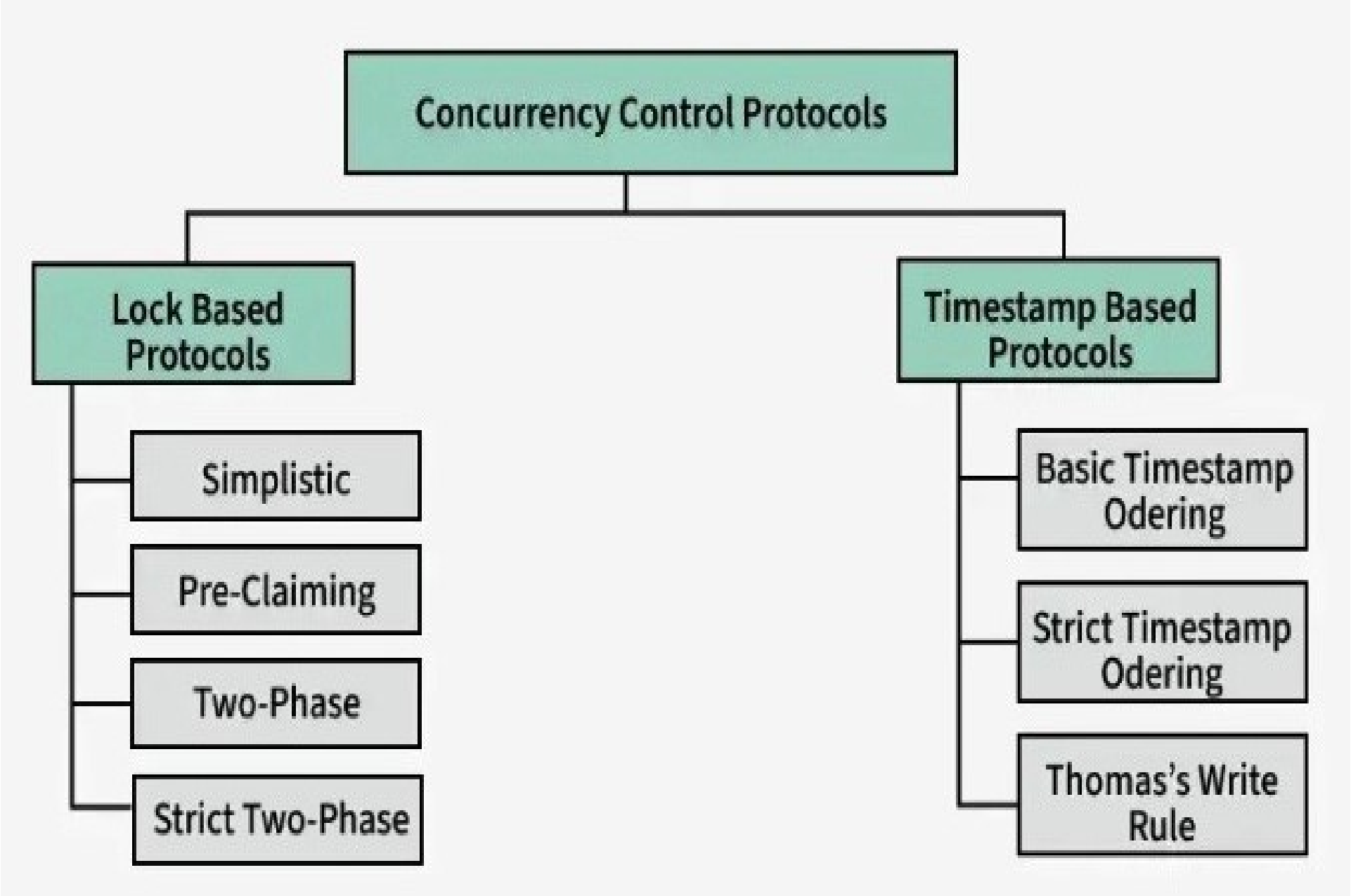
Why Concurrency Control is Needed

Without proper control, running multiple transactions at the same time can cause data errors. Common problems include:

- **Lost Updates:** One transaction overwrites changes made by another concurrent transaction.
- **Dirty Reads:** A transaction reads uncommitted data from another transaction that later aborts or rolls back.
- **Inconsistent Summaries:** A transaction reads partial updates while other transactions are modifying multiple data items.

Scenario	Without Concurrency Control	With Concurrency Control
Initial Balance	$1000	$1000
Transaction A (User 1)	Withdraws $200: Reads balance as $1000, calculates new balance $800.	Withdraws $200: Reads balance as $1000, calculates new balance $800.
Transaction B (User 2)	Withdraws $300: Reads balance as $1000 (before A updates), calculates $ 700	Waits for Transaction A to finish. Once A commits, reads updated balance $800, calculates new balance $500.
Final Updates	A writes $800. Then, B reads overwrites with $700 (lost update issue).	A writes $800. Then, B reads updated balance and writes $500 (consistent result).
Final Balance	$700 (incorrect, inconsistent).	$500 (correct, consistent).

Concurrency Control Protocols



Lock Based protocols

- A lock in DBMS controls concurrent access, allowing only one transaction to use a data item at a time. This ensures data integrity and prevents issues like lost updates or dirty reads during simultaneous transactions.
- Lock Based Protocols in DBMS ensure that a transaction cannot read or write data until it gets the necessary lock. Here's how they work:

➢ These protocols prevent concurrency issues by allowing only one transaction to access a specific data item at a time.

➢ Locks help multiple transactions work together smoothly by managing access to the database items.

➢ Locking is a common method used to maintain the serializability of transactions.

➢ A transaction must acquire a read lock or write lock on a data item before performing any read or write operations on it.

Types of Lock

- **Shared Lock (S):** Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
- **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item.

  X-lock is requested using lock-X instruction.

Rules of Locking

- The basic rules for Locking are given below:

Read Lock (or) Shared Lock(S)

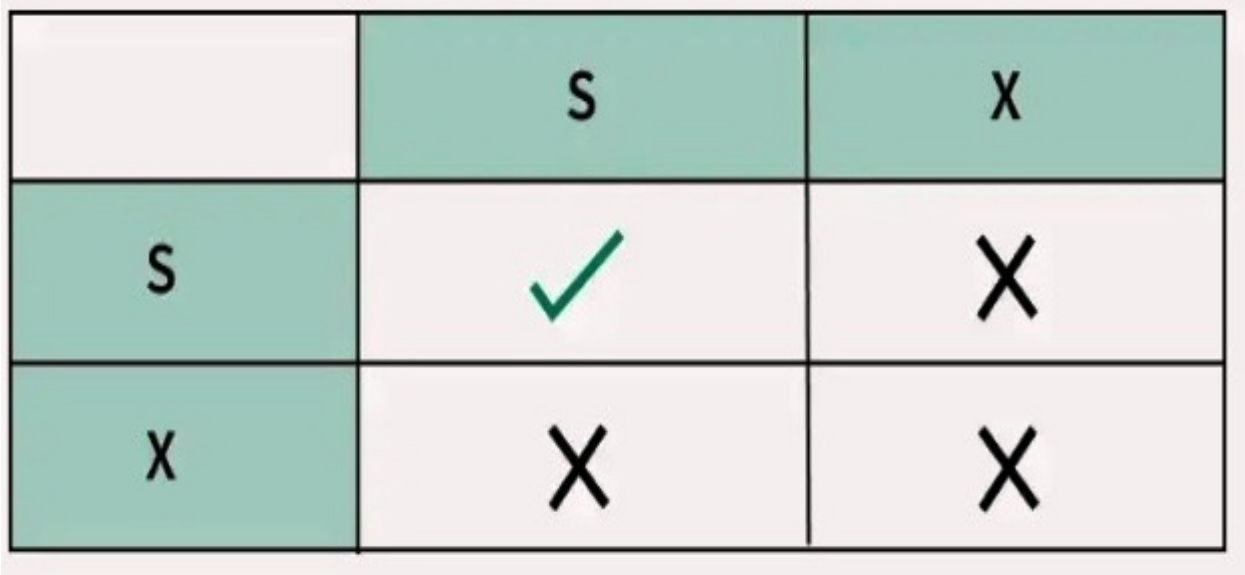
- If a Transaction has a Read lock on a data item, it can read the item but not update it.
- If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.
- So, the Read Lock is also called a Shared Lock.

Write Lock (or) Exclusive Lock (X)

- If a transaction has a write Lock on a data item, it can both read and update the data item.
- If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.
- So, the Write Lock is also known as Exclusive Lock.

Lock Compatibility Matrix

- A transaction can acquire a lock on a data item only if the requested lock is compatible with existing locks held by other transactions.
- **Shared Locks (S):** Multiple transactions can hold shared locks on the same data item simultaneously.
- **Exclusive Lock (X):** If a transaction holds an exclusive lock on a data item, no other transaction can hold any type of lock on that item.
- If a requested lock is not compatible, the requesting transaction must wait until all incompatible locks are released by other transactions.
- Once the incompatible locks are released, the requested lock is granted.



Types of Lock-Based Protocols

1. Simplistic Lock Protocol

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable all transactions to obtain a lock on the data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

Example: Consider a database with a single data item X = 10.

Transactions:

- **T1**: Wants to read and update X.
- **T2**: Wants to read X.

Steps:

1. T1 requests an exclusive lock on X to update its value. The lock is granted.
2. T1 reads X = 10 and updates it to X = 20.
3. T2 requests a shared lock on X to read its value. Since T1 is holding an exclusive lock, T2 must wait.
4. T1 completes its operation and releases the lock.
5. T2 now gets the shared lock and reads the updated value X = 20.

This example shows how simplistic lock protocols handle concurrency but do not prevent problems like deadlocks or limits concurrency.

2. Pre-Claiming Lock Protocol

The Pre-Claiming Lock Protocol avoids deadlocks by requiring a transaction to request all needed locks before it starts. It runs only if all locks are granted; otherwise, it waits or rolls back.

Example: Consider two transactions T1 and T2 and two data items, X and Y:

Transaction T1 declares that it needs:

- A write lock on X.
- A read lock on Y.

Since both locks are available, the system grants them. T1 starts execution: It updates X. It reads the value of Y.

While T1 is executing, Transaction T2 declares that it needs: However, since T1 already holds a write lock on X, T2's request is denied. T2 must wait until T1 completes its operations and releases the locks.
A read lock on X

Once T1 finishes, it releases the locks on X and Y. The system now grants the read lock on X to T2, allowing it to proceed.
This method is simple but may lead to inefficiency in systems with a high number of transactions.

3. Two-phase locking (2PL)

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases :

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

4. Strict Two-Phase Locking Protocol

Strict Two-Phase Locking requires that in addition to the 2-PL all Exclusive(X) locks held by the transaction be released until after the Transaction Commits.

5. Rigorous Two- Phase Lock Protocol

A stricter form of the Two-Phase Locking (2PL) protocol where all locks (both Shared and Exclusive) are held until the transaction commits.

Timestamp based Concurrency Control

- Timestamp-based concurrency control is a technique used in database management systems (DBMS) to ensure serializability of transactions without using locks. It uses timestamps to determine the order of transaction execution and ensures that conflicting operations follow a consistent order.
- Each transaction T is assigned a unique timestamp TS(T) when it enters the system. This timestamp determines the transaction's place in the execution order.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol enforces that older transactions (with smaller timestamps) are given higher priority. This prevents conflicts and ensures the execution is serializable and deadlock-free.
- **For example:**
- If Transaction T1 enters the system first, it gets a timestamp TS(T1) = 007 (assumption).
- If Transaction T2 enters after T1, it gets a timestamp TS(T2) = 009 (assumption).
- This means T1 is "older" than T2 and T1 should execute before T2 to maintain consistency.

Features of Timestamp Ordering Protocol:

1. Transaction Priority:

- Older transactions (those with smaller timestamps) are given higher priority.
- For example, if transaction T1 has a timestamp of 007 times and transaction T2 has a timestamp of 009 times, T1 will execute first as it entered the system earlier.

2. Early Conflict Management:

Unlike lock-based protocols, which manage conflicts during execution, timestamp-based protocols start managing conflicts as soon as a transaction is created.

3. Ensuring Serializability:

The protocol ensures that the schedule of transactions is serializable. This means the transactions can be executed in an order that is logically equivalent to their timestamp order.

Core Concepts

- **Transaction Timestamp (TS(T)):** A unique, monotonically increasing identifier assigned to each transaction (T) when it enters the system (using a system clock or a logical counter).
    - *Older transaction:* Smaller TS value (higher priority).
    - *Younger transaction:* Larger TS value (lower priority).
- **Read Timestamp ($R_TS(X)$):** The largest timestamp of any transaction that has successfully read data item X.
- **Write Timestamp ($W_TS(X)$):** The largest timestamp of any transaction that has successfully written to data item X.

Basic Timestamp Ordering Rules

When a transaction T (with timestamp TS(T)) attempts to perform an operation on a data item X, the DBMS checks the operation against the data item's $R_TS(X)$ and $W_TS(X)$.

1. Read Operation: Read(X)

- If $TS(T) < W_TS(X)$:
  - This means a *younger* transaction has already overwritten the value that transaction T is trying to read.
  - **Action:** T is **aborted and rolled back**, then restarted with a new, larger timestamp.
- If $TS(T) \geq W_TS(X)$:
  - The read is **allowed** to execute.
  - Update: $R_TS(X) = \max(R_TS(X), TS(T))$.

2. Write Operation: Write(X)

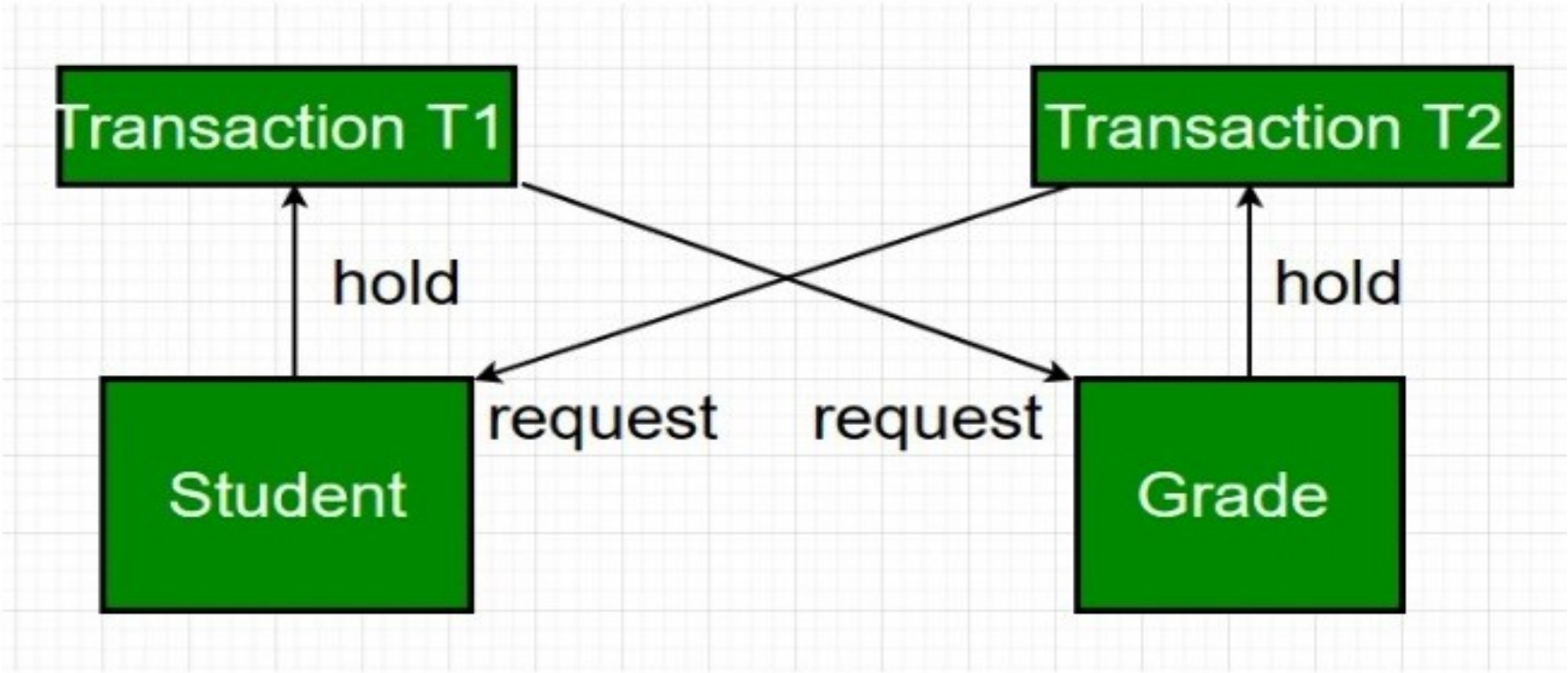
- If $TS(T) < R_TS(X)$:
  - This means a *younger* transaction has already read the value of X, and writing now would violate timestamp order.
  - **Action:** T is **aborted and rolled back**.
- If $TS(T) < W_TS(X)$:
  - This means a *younger* transaction has already updated data item X (handled via **Thomas's Write Rule** below, or aborted in basic protocols).
  - **Action:** T is **aborted and rolled back** (or the write is skipped under Thomas's Write Rule).
- Otherwise ($TS(T) \geq R_TS(X)$ and $TS(T) \geq W_TS(X)$):
  - The write is **allowed** to execute.
  - Update: $W_TS(X) = TS(T)$

Deadlock in DBMS

- A deadlock occurs in a DBMS when two or more transactions are waiting for each other to release resources, creating a cycle where none of the transactions can proceed.
- This creates an indefinite wait, which blocks further execution. These DBMS deadlocks have to be handled properly to ensure efficient database operations.

Deadlock in DBMS

For Example: Consider the image below



Deadlock in DBMS

In the above image, we can see that:

- T1 locks Resource "Student" and needs Resource "Grade"
- T2 locks Resource "Grade" and needs Resource "Student"
- T1 waits for T2, T2 waits for T1, hence resulting in a deadlock

Necessary Conditions of Deadlock

- For a deadlock to occur, all four of these conditions must be true:
- **Mutual Exclusion:** Only one transaction can hold a particular resource at a time.
- **Hold and Wait:** The Transactions holding resources may request additional resources held by others.
- **No Preemption:** The Resources cannot be forcibly taken from the transaction holding them.
- **Circular Wait:** A cycle of transactions exists where each transaction is waiting for the resource held by the next transaction in the cycle.

Deadlock Handling in DBMS

- DBMS deadlock handling ensures that transactions do not remain indefinitely blocked in case of a circular wait condition. Deadlock handling is done by prevention, detection, and recovery mechanisms to keep the system efficient and data consistent.
- **Deadlock Avoidance**
- It is a technique that ensures that the system never gets into a deadlock state by carefully allocating resources only if it guarantees a safe execution path. Algorithms, like Banker's Algorithm, can be used to prevent deadlocks before they occur.

Deadlock Handling in DBMS

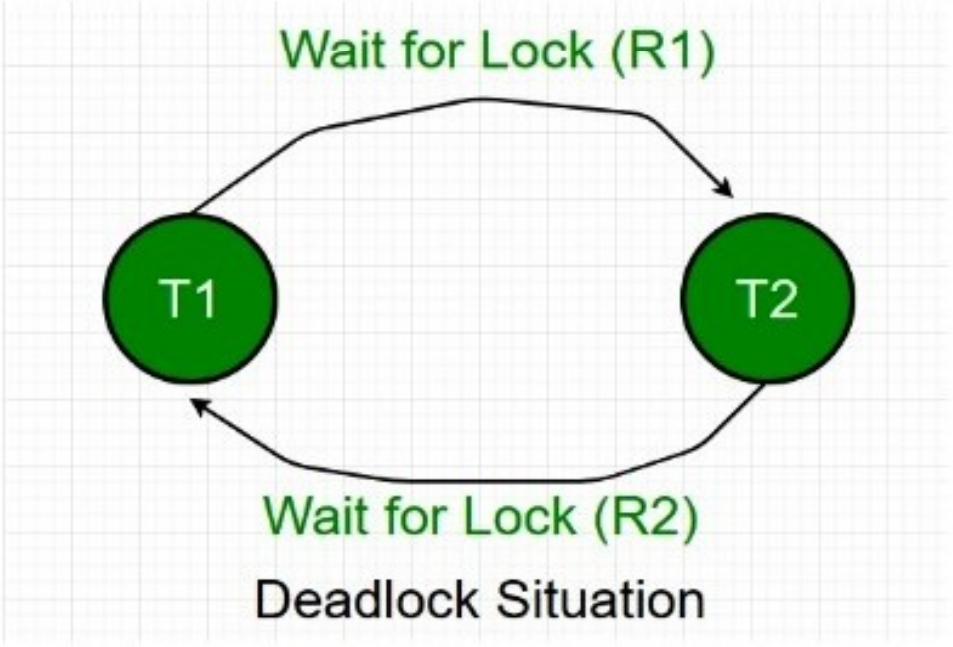
- **Deadlock Detection**
- Deadlock detection is a procedure adopted to discover DBMS deadlocks and fix them after their occurrence.
- Deadlock detection differs from deadlock prevention and deadlock avoidance, which resolve the situation before it arises.
- Once a deadlock occurs, deadlock detection can be used to cycle detection algorithms on the Resource Allocation Graph (RAG) or wait-for graphs to determine whether or not a deadlock exists.
- Once a deadlock is detected, some corrective action is initiated by the DBMS, either by terminating the transactions and releasing the resources or by some form of resource preemption to break the deadlock and restore normal operation of the DBMS.

Deadlock Handling in DBMS

- **Deadlock Detection**

Method: Wait-For Graph

- Nodes: Transactions
- Edges: Waiting relationships
- If there's a cycle, a deadlock exists. It's mostly suitable for small to medium databases



Deadlock Handling in DBMS

- **Deadlock Prevention**
- By removing at least one of the four conditions of deadlock (Mutual Exclusion, Hold and Wait, No Preemption, and Circular Wait), it can be prevented.
- Deadlock prevention is a DBMS technique that guarantees DBMS deadlocks never happen. Prevention techniques like Wait-die and Wound-wait can be used.

Deadlock Handling in DBMS

- **Deadlock Prevention**

Wait-Die Scheme

Wait-Die scheme is a deadlock prevention technique based on the timestamp of the transaction. While allowing only older transactions to wait and forcing younger transactions to abort(die) and restart ensures that DBMS deadlocks never happen.

How does the Wait-Die Scheme work?

Step 1: Assign a timestamp to each transaction

- Based on what time the transaction began, the system assigns it a timestamp (TS).
- Older transactions have smaller timestamps (lower values).
- Younger transactions have larger timestamps (higher values).

Step 2: Transaction requests

When a transaction requests a resource that is already locked by another transaction, then these rules can be followed:

- If the requesting transaction has a lower timestamp, i.e older transaction, then it waits.
- If the requesting transaction has a higher timestamp, i.e younger transaction, it aborts(dies) and restarts later.

Deadlock Handling in DBMS

- **Deadlock Prevention**

Wound-Wait Scheme

The Wound-Wait scheme is a deadlock prevention method that uses timestamps to determine which transaction waits and which aborts. In contrast to Wait-Die, the Wound-Wait scheme allows younger transactions to wait, while older transactions are allowed to wound (abort), if necessary.

How does the Wound-wait scheme work?

Step 1: Assigning a timestamp to each transaction

- Based on what time the transaction began, the system assigns it a timestamp (TS).
- Older transactions have smaller timestamps (lower values).
- Younger transactions have larger timestamps (higher values).

Step 2: Transaction Request

When a transaction requests a resource that is already locked by another transaction, then these rules can be followed:

- If the request comes from an older transaction, it makes the newer one stop and fail.
- If the requesting transaction has a higher timestamp, i.e younger transaction, it waits until the resource is available.