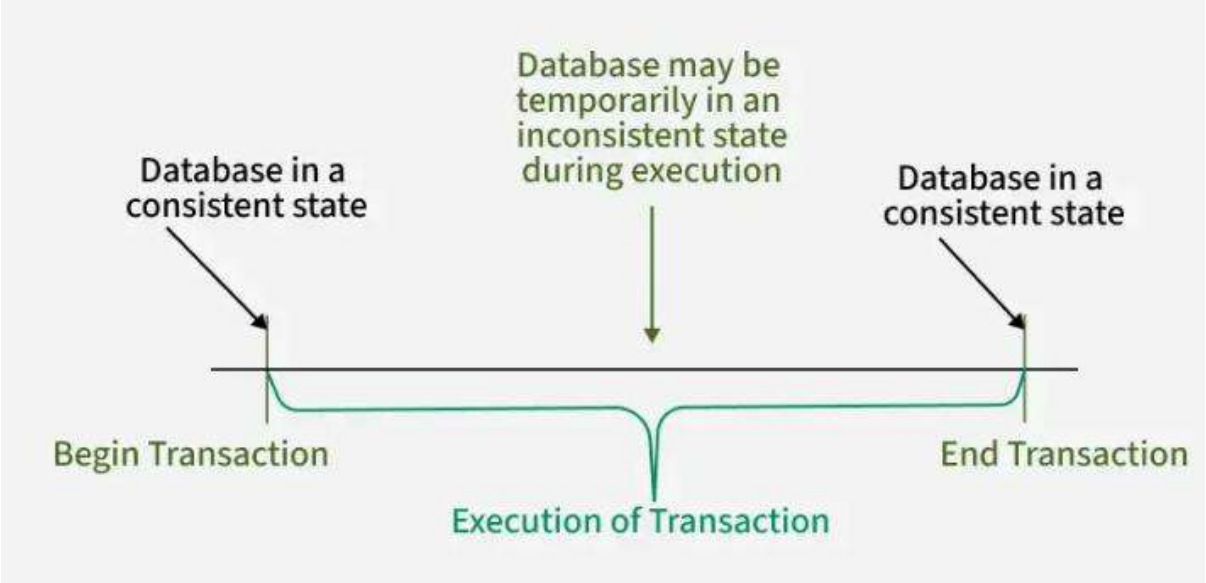
UNIT -4 Transaction and Concurrency Control

Transaction in DBMS

A transaction refers to a sequence of one or more operations (such as read, write, update, or delete) **performed on the database as a single logical unit of work.**

- A transaction **ensures** that either all the operations are successfully **executed** (committed) or **none** of them take effect (rolled back).
- Transactions are designed to **maintain** the **integrity**, **consistency** and **reliability** of the database, even in the **case of system failures** or concurrent access.



Transaction

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction.

During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

Example: Let's consider an **online banking application**:

Transaction: When a user performs a **money transfer**, several operations occur, such as:

- **Reading** the account balance of the sender.
- **Writing** the deducted amount from the sender's account.
- **Writing** the added amount to the recipient's account.

Operations of Transaction

A user can make **different types of requests** to **access** and **modify** the **contents of a database**. So, we have **different types of operations** relating to a transaction. They are discussed as follows:

1) Read(X)

A **read operation** is used to **read the value of a particular database element X** and **stores it in a temporary buffer** in the main memory for further actions such as displaying that value.

Example: For a banking system, when a user checks their balance, a Read operation is performed on their **account** balance:

SELECT balance FROM accounts WHERE account_id = 'A123';

This updates the balance of the user's account after withdrawal.

2) Write(X)

A write operation **stores updated data from main memory back to the database**.
It usually **follows a read**, where data is fetched, modified (e.g., arithmetic changes) and then **written back** to save the updated value.

Example: For the banking system, if a user withdraws money, a **Write** operation is performed after the balance is updated:

UPDATE accounts SET balance = balance - 100 WHERE account_id = 'A123';

This updates the balance of the user's account after withdrawal.

3) Commit

This operation in transactions is **used to maintain integrity in the database**. Due to some failure of power, hardware, or software, etc., **a transaction might get interrupted before all its operations are completed**.
This may cause ambiguity in the database, i.e. it might get **inconsistent** before and after the transaction.

Example: After a successful money transfer in a banking system, a **Commit** operation finalizes the transaction:
COMMIT;

Once the transaction is committed, the changes to the database are permanent and the transaction is considered **successful**.

4) Rollback

A rollback **undoes all changes made by a transaction** if an error occurs, restoring the database to its last consistent state.

It helps prevent **data inconsistency** and **ensures safety**.

Example: Suppose during the money transfer process, the system **encounters an issue**, like **insufficient funds** in the sender's account. In that case, the transaction is rolled back:
ROLLBACK;

This will undo all the operations performed so far and ensure that the database remains consistent.

ACID Properties of Transaction

Transactions in DBMS must **ensure** data is **accurate** and **reliable**. They follow **four key ACID properties**:

1. **Atomicity:** A transaction is all or nothing. If any part fails, the entire transaction is rolled back.
   For example, when transferring money, both the debit and credit operations must succeed; if either one fails, no changes should occur.

2. **Consistency:** A transaction **must keep the database in a valid state**, moving it from one consistent state to another.
   Example: If balance is ₹1000 and ₹200 is withdrawn, the new balance should be ₹800.

3. **Isolation:** Transactions run independently. One transaction's operations should not affect another's intermediate steps.
   Example: Two users withdrawing from the same account must not interfere with each other's balance updates.

4. **Durability:** Once a transaction is committed, its changes stay even if the system crashes.

Example: After a successful transfer, the updated balance remains safe despite a power failure.

Transaction States

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.

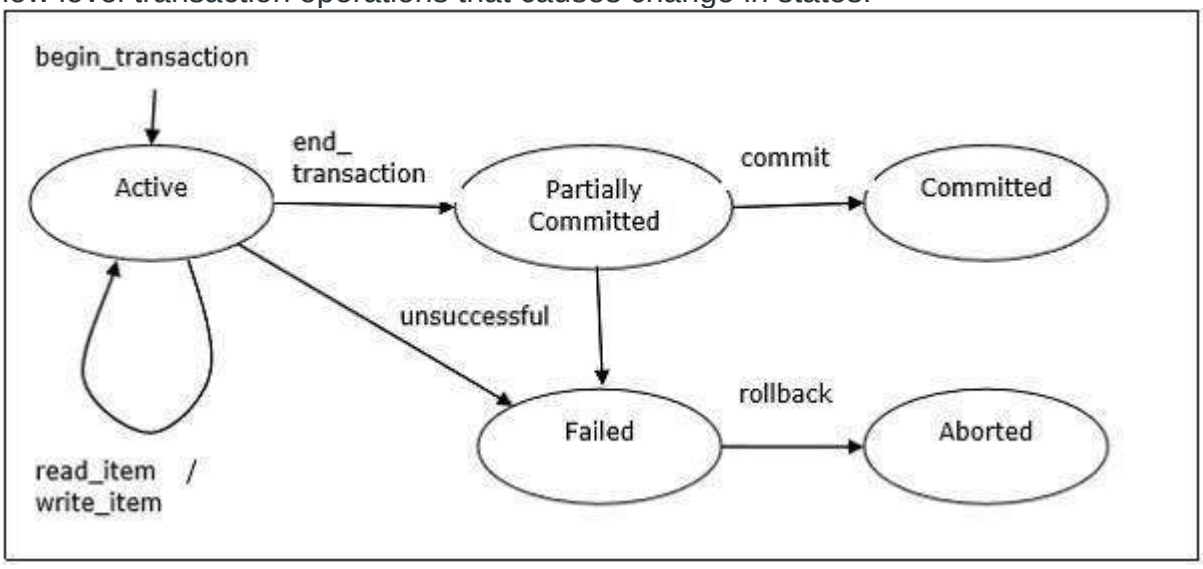
- **Partially Committed** − The transaction enters this state **after the last statement of the transaction has been executed**.

- **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.**

The following state transition diagram depicts the states in the transaction and the low level transaction operations that causes change in states.



Transaction Schedules

When **multiple transaction requests are made at the same time**, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions.

There are broadly **two types** of transaction schedules discussed as follows:

i) Serial Schedule

In a serial schedule, **transactions execute one at a time**, ensuring database consistency **but increasing waiting time** and **reducing system throughput.**
To improve throughput while maintaining consistency, concurrent schedules with strict rules are used, allowing safe simultaneous execution of transactions.

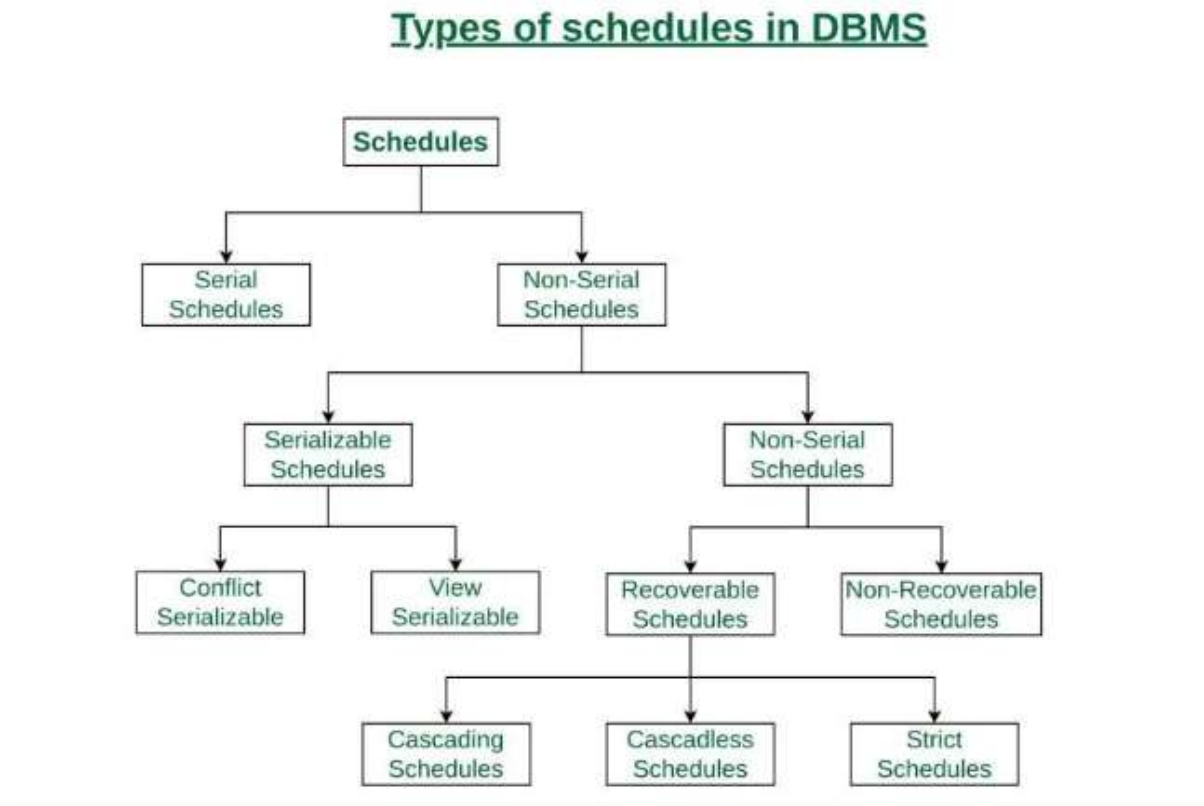
ii) Non-Serial Schedule

Non-serial schedule is a type of transaction schedule where **multiple transactions are executed concurrently**, interleaving their operations, instead of running one after another. It improves system efficiency **but requires concurrency control** to maintain database consistency.

Types of Schedules in DBMS

Scheduling is the process of determining the order in which transactions are executed.

When **multiple transactions run concurrently**, **scheduling ensures** that operations are executed in a way **that prevents conflicts or overlaps** between them.
There are several types of schedules, all of them are depicted in the diagram below:



Let's discuss the different types of schedules one by one:

Serial Schedule

Schedules in which the transactions are **executed non-interleaved**, i.e., a serial schedule is one in which **no transaction starts until a running transaction has ended** are called serial schedules.

Example: Consider the following schedule involving two transactions T_1 and T_2 .

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

T_1	T_2
	R(A)
	R(B)

where **R(A) denotes that a read operation is performed on some data item 'A'** This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

Non-Serial Schedule

This is a type of Scheduling where the operations of **multiple transactions are interleaved**.
This might lead to a **rise in the concurrency problem**.

The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule.
This sort of **schedule does not provide any benefit of the concurrent transaction.**
The schedule runs transactions in an interleaved (non-serial) order, but **behaves exactly like a serial order**. Since there is **no real parallel execution** advantage, concurrency gives no performance benefit.

It can be of two types namely,

1.Serializable Scheduling (Concurrency Control)

- **Ensures** database **consistency in non-serial schedules** by verifying if they **behave like serial schedules**.
- In a serial schedule, transactions execute one after another, ensuring correctness without conflicts.
  - **Its final result is the same as some serial** (one-after-another) **execution** of the same transactions.
  - Even though the operations may be interleaved (executed concurrently), the **outcome must match a schedule where transactions run without overlapping**.

Why to use Serializable?

When **many users access a database at once**:

- Their transactions may interact
- They may read or write the same data

- This can cause incorrect results (lost updates, dirty reads, etc.)
- Serializable schedules **prevent these problems** and guarantee data correctness.
- Allows **better CPU** and **resource utilization**.
- Improves **throughput** without sacrificing consistency.

Two types of serializable scheduling are:

i. Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule **by swapping non-conflicting operations**.

Two operations are **said to be conflicting if all conditions satisfy**:
- They **belong to different transactions**
- They **operate on the same data item**
- **At Least one** of them is a **write operation**

ii. View Serializable: A Schedule is called view serializable if it is view equal to a serial schedule (**no overlapping transactions**).
A conflict-serializable schedule is always view-serializable, but a view-serializable schedule that contains **blind writes** may not be conflict-serializable.
- Every schedule that is **conflict-serializable** is **also view-serializable**.
- Conflict serializability is a **stricter** (more restrictive) condition.
- So:
  Conflict-serializable ⊆ View-serializable

If a **schedule passes the *conflict* test**, it **automatically passes the *view* test.**

A **blind write** is when a transaction **writes a value without reading it first**.
Example:

W(A) without any R(A) before it in that transaction.

2. Non-Serializable Scheduling

Schedules that **do not preserve serial equivalence** and may lead to inconsistencies if not handled carefully.

Two schedules are serially equivalent if:

They produce the same final database state, and Each read sees the same value it would in the equivalent serial schedule.

i. Recoverable Schedule: A recoverable schedule is a schedule in which a **transaction is allowed to commit only** after **all the transactions from which it has read data have committed.**

In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

There can **be three types of recoverable** schedule:

ii. Cascading Schedule: Also called **Avoids cascading aborts/rollbacks (ACA)**.

When there is a **failure in one transaction** and this **leads to the rolling back or aborting other dependent transactions**, then such scheduling is referred to as Cascading rollback or cascading abort.

A cascading abort (or cascading rollback) occurs when:

One transaction aborts, and because other transactions have **read its uncommitted (dirty) data**, those other transactions must also be aborted.

This creates a chain reaction of aborts, like falling dominoes.
Example:

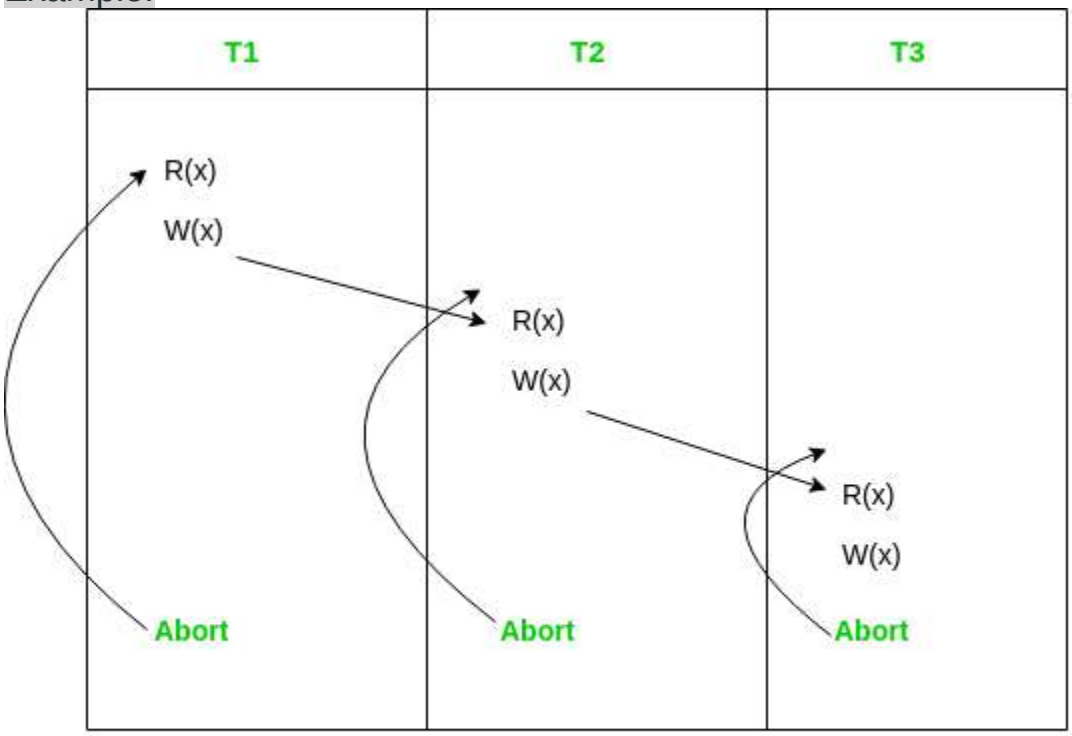


Figure - Cascading Abort

iii. Cascadeless Schedule: Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules.

A **cascadeless schedule (also called ACA: Avoids Cascading Aborts**) is a schedule in which:

A **transaction is allowed to read a data item only** after the **transaction that last wrote that item has committed.**

In simple words:

No transaction is allowed to read uncommitted data.

Every read operation happens only after the **writer transaction commits**.

Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i , then the commit of T_j must read it after the commit of T_i .
Example: Consider the following schedule involving two transactions T_1 and T_2 .

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Example: Consider the following schedule involving two transactions T_1 and T_2 .

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as **T_2 has already read the uncommitted** value written by T_1 .

iv. Strict Schedule:

A schedule is strict if it obeys the following rule:

If Transaction Ti writes a data item first, then no other transaction Tj is allowed to read or write that same item until Ti has either committed or aborted.

In other words:

No transaction may access a data item that was written by another transaction until that writer finishes (commit/abort).

This is the **strongest form of scheduling** used in databases.

Example: Consider the following schedule involving two transactions T_1 and T_2 .

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1 .

v. Non-Recoverable Schedule:

A non-recoverable schedule is a schedule in which:

A **transaction commits** even though it has **read data written by another transaction that has not yet committed**.

If the **writer transaction later aborts**, the **reader has already committed** using **dirty (uncommitted) data**, and the system cannot fix the inconsistency.

This makes the schedule non-recoverable, meaning the **database cannot safely recover from an abort.**

Example: Consider the following schedule involving two transactions T_1 and T_2 .

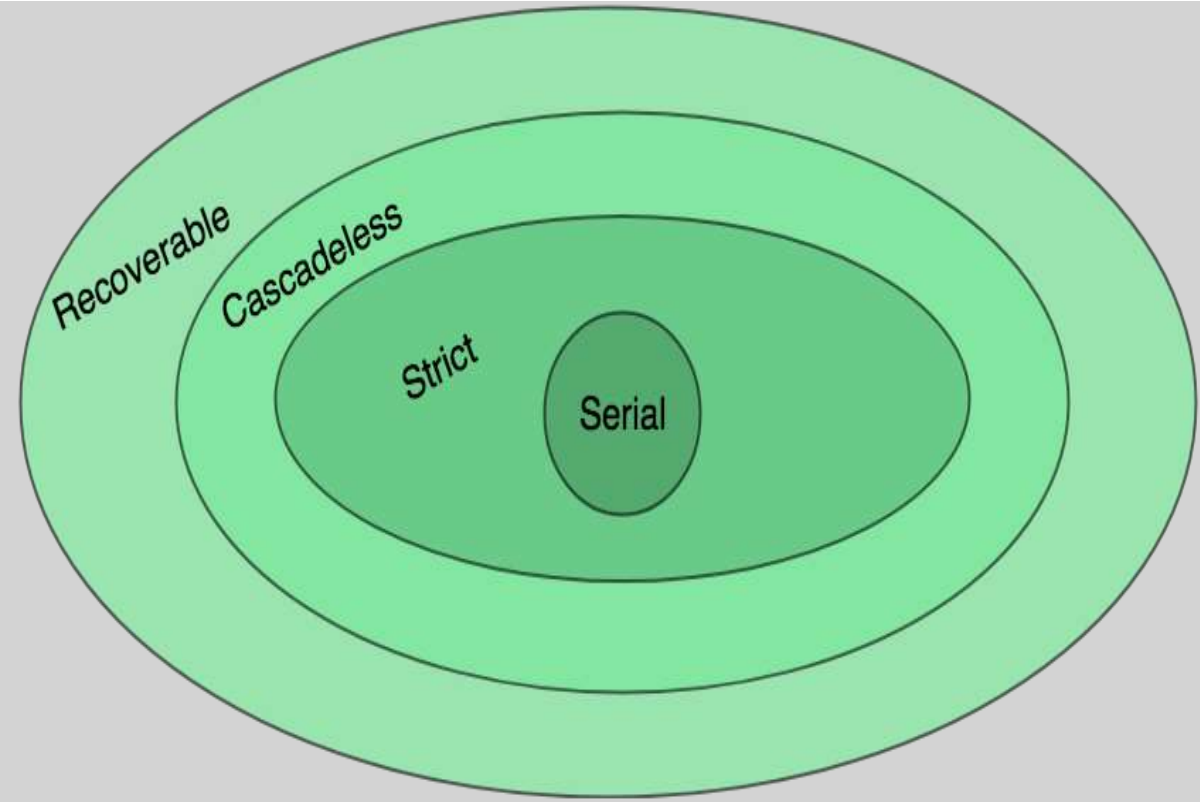
T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, **but since T_2 committed, this schedule is non-recoverable.**

Note - It can be seen that:

- Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
- Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
- Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The **relation between various types of schedules** can be depicted as:



Example:

Consider the following schedule:
S: R1(A), W2(A), Commit2, W1(A), W3(A), Commit3, Commit1

Question: Which of the following is true?
(A) The schedule is view serializable schedule and strict recoverable schedule(
B) The schedule is non-serializable schedule and strict recoverable schedule
(C) The schedule is non-serializable schedule and is not strict recoverable schedule.
(D) The Schedule is serializable schedule and is not strict recoverable schedule

Solution: The schedule can be re-written as:-

T_1	T_2	T_3
R(A)		
	W(A)	

T_1	T_2	T_3
	Commit	
W(A)		
		W(A)
		Commit
Commit		

First of all, it is a view serializable schedule as it has view equal serial schedule T_1 —> T_2 —> T_3 which satisfies the initial and updated reads and final write on variable A which is required for view serializability.

Now we can see there is write – write pair done by transactions T_1 followed by T_3 which is violating the above-mentioned condition of strict schedules as T_3 is supposed to do write operation only after T_1 commits which is violated in the given schedule.
Hence the given schedule **is serializable but not strict recoverable**.
So, option (D) is correct.

Precedence Graph

Precedence graph is a graphical representation of transactions and their dependencies.

It helps identify conflicts and determine if a schedule is conflict serializable.

A cycle in the precedence graph indicates a potential conflict and violates serializability.

a) Precedence Graph for Schedule-1

b) Precedence Graph for Schedule-2

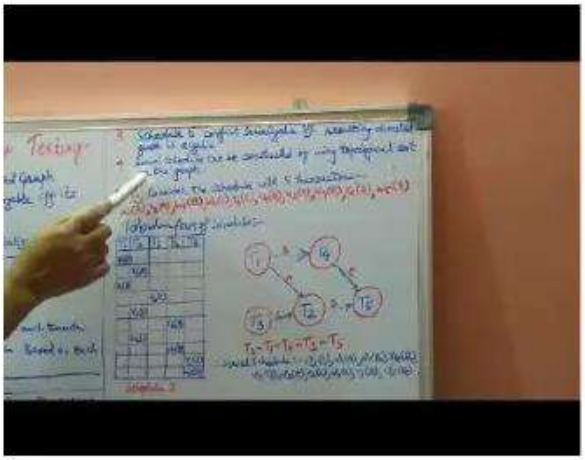
c) Precedence Graph for Schedule-4

Testing for Serializability - Step 1

Step 1: Construct a precedence graph for the given schedule.

Identify transactions and their operations.

Determine dependencies between conflicting operations.

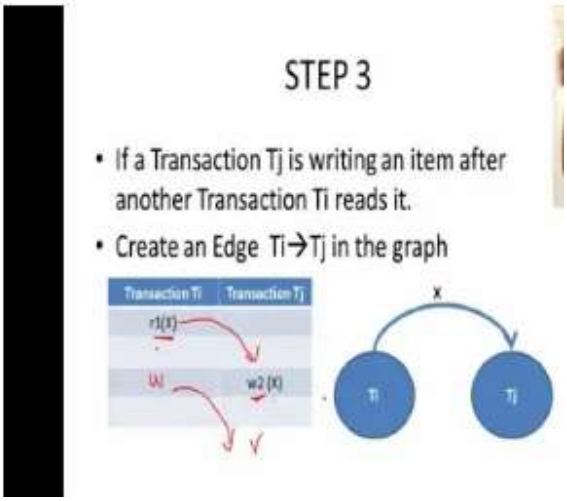


Testing for Serializability - Step 2

Step 2: Check for cycles in the precedence graph.

If a cycle exists, the schedule is not conflict serializable.

Cycles indicate conflicting operations that violate serializability.



Testing for Serializability - Step 3

Step 3: If no cycles exist, the schedule is conflict serializable.

The schedule can be executed in a serial order without conflicts.

Serializability guarantees consistency and data integrity.

Test for Conflict Serializability

- Let S be he schedule. Get the precedence graph G = (V,E).
- V, Vertices are Transactions and E are edges Ti → Tj for
  - Ti Write(Q) before Tj Read(Q)
  - Ti Read(Q) before Tj Write(Q)
  - Ti Write(Q) before Tj Write(Q)
- If there are no cycles in G then it is conflict-serializable.
- Serial schedule is obtained by topological sorting. See examples.

PRACTICE PROBLEMS BASED ON CONFLICT SERIALIZABILITY-

Problem-01:

Check whether the given schedule S is conflict serializable or not-

$$S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)$$

Solution-

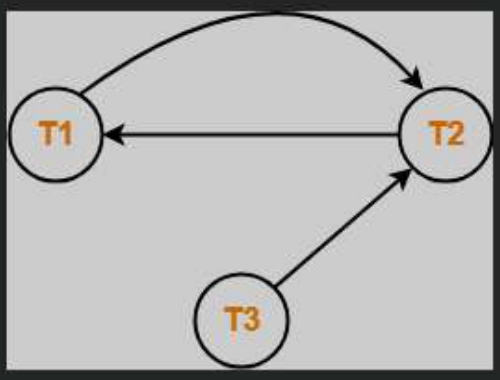
Step-01:

List all the conflicting operations and determine the dependency between the transactions-

- $R_2(A) , W_1(A)$ $(T_2 \rightarrow T_1)$
- $R_1(B) , W_2(B)$ $(T_1 \rightarrow T_2)$
- $R_3(B) , W_2(B)$ $(T_3 \rightarrow T_2)$

Step-02:

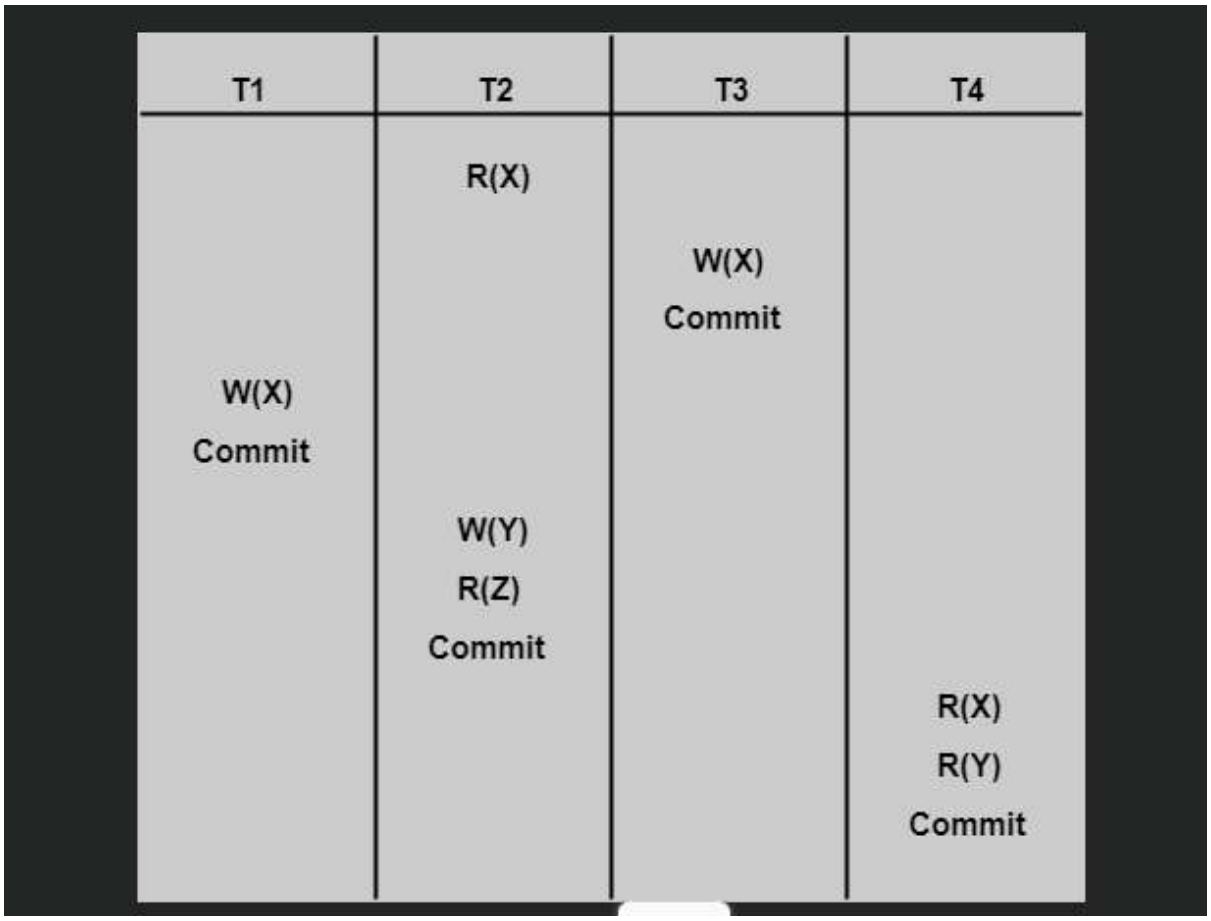
Draw the precedence graph-



- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

Problem-02:

Check whether the given schedule S is conflict serializable and recoverable or not-



T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit

Solution-

Checking Whether S is Conflict Serializable Or Not-

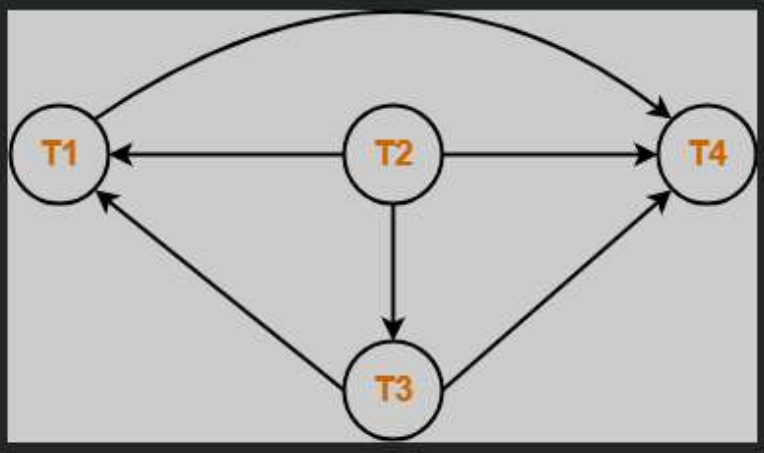
Step-01:

List all the conflicting operations and determine the dependency between the transactions-

- $R_2(X)$, $W_3(X)$ $(T_2 \rightarrow T_3)$
- $R_2(X)$, $W_1(X)$ $(T_2 \rightarrow T_1)$
- $W_3(X)$, $W_1(X)$ $(T_3 \rightarrow T_1)$
- $W_3(X)$, $R_4(X)$ $(T_3 \rightarrow T_4)$
- $W_1(X)$, $R_4(X)$ $(T_1 \rightarrow T_4)$
- $W_2(Y)$, $R_4(Y)$ $(T_2 \rightarrow T_4)$

Step-02:

Draw the precedence graph-



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.

Concurrency Control in DBMS

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

Need of Concurrency Control

Concurrency control is essential to:

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
- Prevent conflicts between simultaneous transactions.
- Maintain data consistency and accuracy in multi-user environments.
- Avoid problems such as dirty reads, lost updates and inconsistent reads.

Example:
- Without concurrency control: Two users update the same record simultaneously and one update overwrites the other.

- With concurrency control: The DBMS uses locks or timestamps to ensure updates occur sequentially and data remains correct.

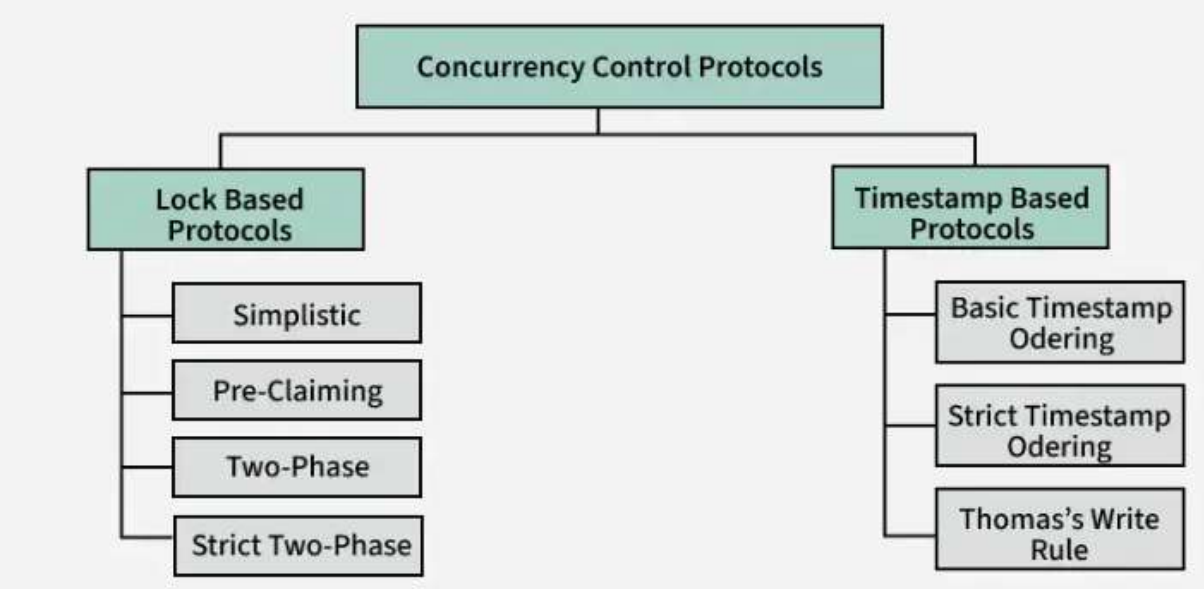
Concurrency Problems in DBMS

When multiple transactions execute concurrently, several problems may occur:

- **Dirty Read:** A transaction **reads uncommitted data** from another transaction that may later roll back.
- **Lost Update:** Two transactions **update the same data** and **one update overwrites the other.**
- **Inconsistent Read:** A transaction **reads the same data multiple times** and the **data changes in between reads**.

Concurrency Control Protocols

Concurrency control protocols define rules to ensure correct and consistent execution of transactions. The main protocols are:



Concurrency Control Protocols

1. Lock-Based Concurrency Control:

- **Uses locks to restrict access to data items** during a transaction. **Common types** include **shared locks** (read) and **exclusive locks** (write).
- Ensures serializability and prevents conflicts.
- **Example:** Two-Phase Locking (2PL) guarantees that once a transaction releases a lock, it cannot obtain any new locks.

2. Timestamp-Based Concurrency Control:

- Each transaction is **assigned a timestamp**.

- The DBMS uses these timestamps to order transactions and prevent conflicts based on their start time.

Advantages of Concurrency Control

1. **Reduced Waiting Time:** Multiple transactions can proceed simultaneously, reducing idle time.
2. **Improved Response Time:** Faster access and interaction with the database.
3. **Better Resource Utilization:** Hardware and database resources are efficiently shared.
4. **Increased System Efficiency:** Higher throughput and better overall performance.

Disadvantages of Concurrency Control

1. **Overhead:** Managing locks and timestamps **adds system overhead.**
2. **Deadlocks:** Circular waits between transactions can halt progress.
3. **Reduced Concurrency:** Locking can limit the number of simultaneous transactions.
4. **Complexity:** Implementation in distributed or large systems can be difficult.
5. **Inconsistencies:** Rollbacks or long waits may cause temporary data inaccuracy or staleness.

Concurrency problems in DBMS Transactions

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in a database environment.

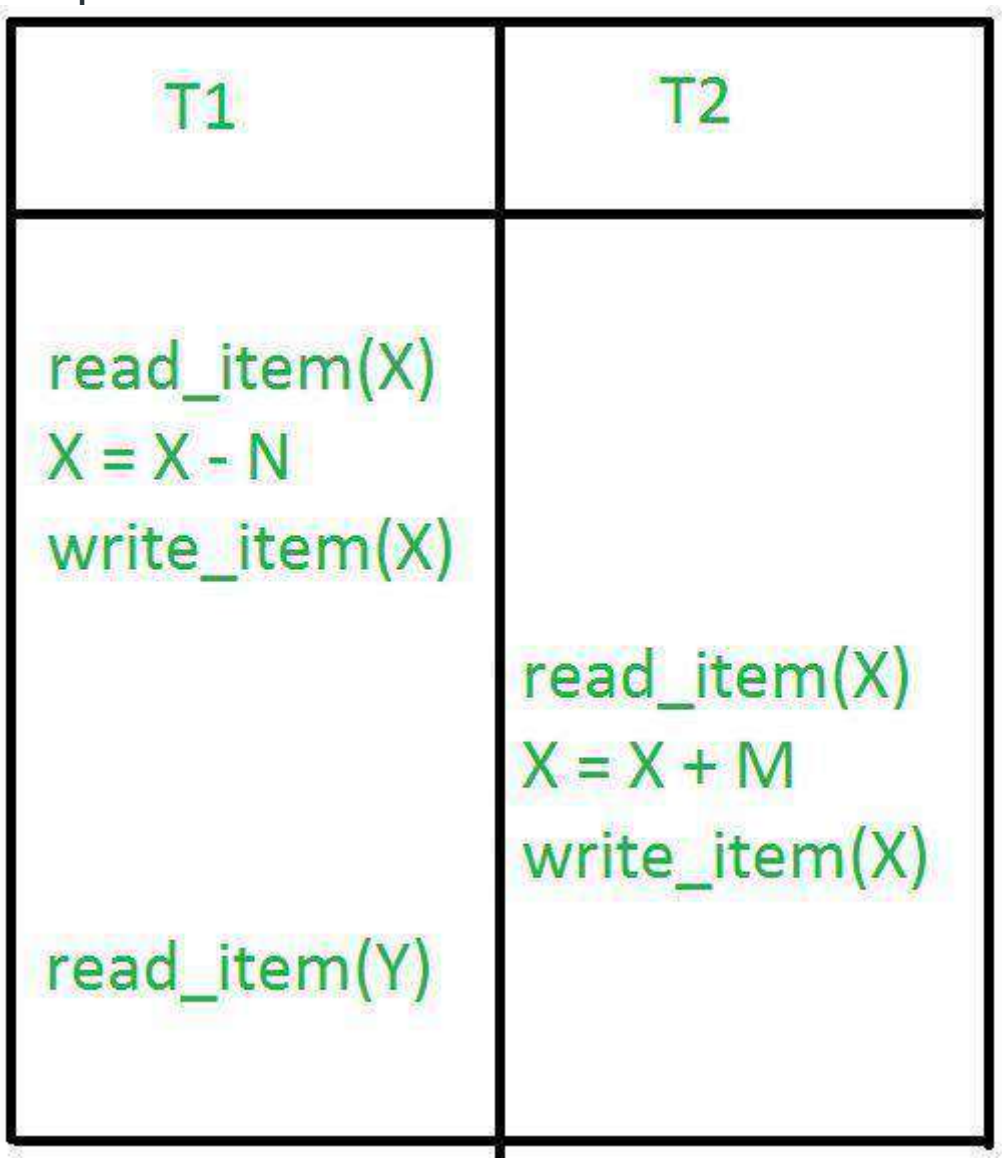
The **five concurrency problems that can occur in the database are**:

- Temporary Update Problem
- Incorrect Summary Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem

Temporary Update Problem:

Temporary update or dirty read problem occurs **when one transaction updates an item and fails.** But the **updated item is used by another transaction** before the item is changed or reverted back to its last value.

Example:



In the above **example**, **if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.**

Incorrect Summary Problem:

Consider a situation, where **one transaction is applying the aggregate function on some records** while another transaction is updating these

records. The **aggregate function may calculate some values before the values have been updated** and others after they are updated.

The incorrect summary problem occurs when one transaction is calculating an aggregate (SUM, AVG, COUNT, etc.) on a set of records while another transaction is concurrently updating those same records.

Because the reads and updates interleave, part of the summary is computed using old values and part using new values, giving an incorrect final result.

Example Scenario

Initial Table: Accounts

Account	Balance
A1	100
A2	150
A3	250

Transaction T1:

Calculates **total balance** of all accounts.

Transaction T2:

Updates balances (e.g., adds ₹50 to each account).

Timeline (Interleaving)

Step	Transaction T1 (Summary)	Transaction T2 (Update)
1	Reads A1 = 100	
2		Updates A1 → 150
3	Reads A2 = 150	
4	Reads A3 = 250	
5		Updates A2 → 200
6		Updates A3 → 300

Why the problem occurs

Because the summary transaction T1:

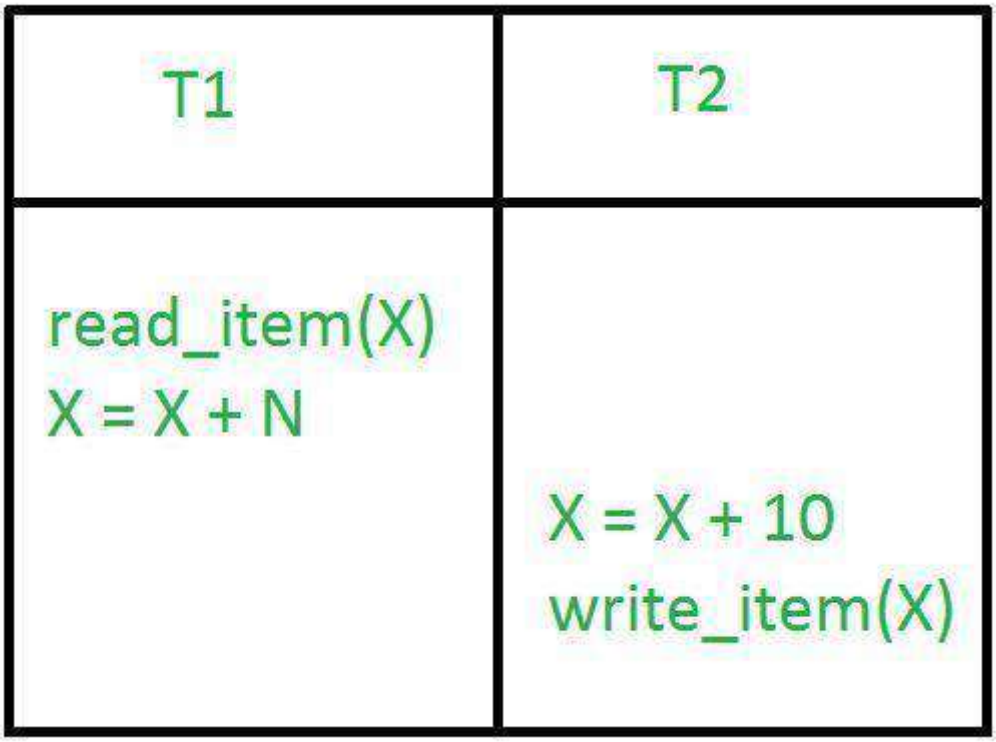
- Reads some rows **before** they are updated, and
- Reads others **after** they are updated

→ Leading to an incorrect, inconsistent summary.

Lost Update Problem:

In the lost update problem, **an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction**.

Example:

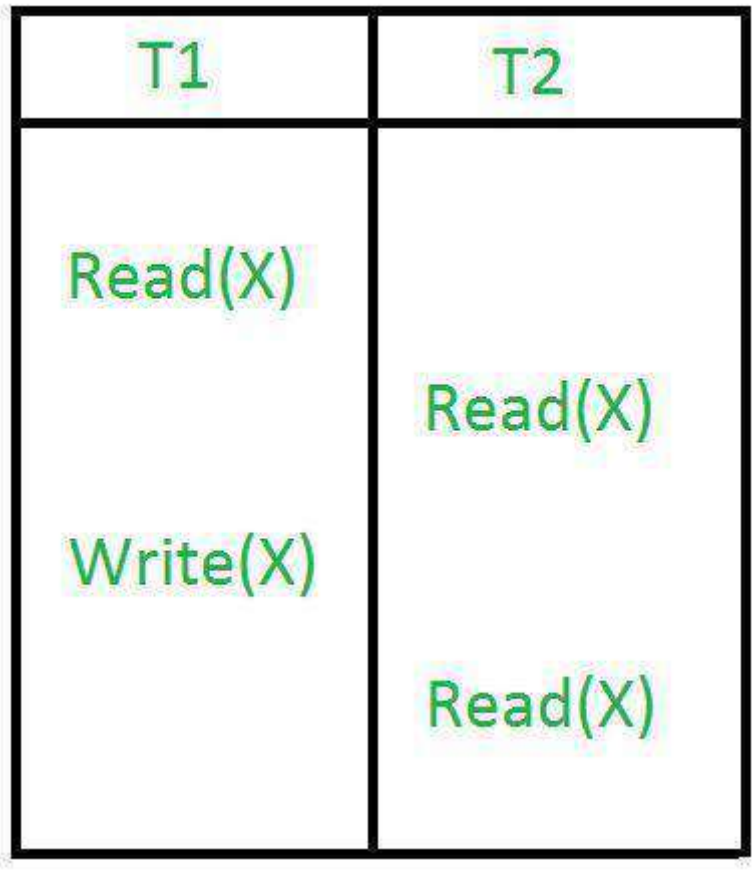


In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*.

Therefore, the update done by transaction 2 will be lost. Basically, **the write commit done by the last transaction will overwrite all previous write commits.**

Unrepeatable Read Problem:

The unrepeatable problem occurs **when two or more read operations of the same transaction read different values of the same variable**.
Example:

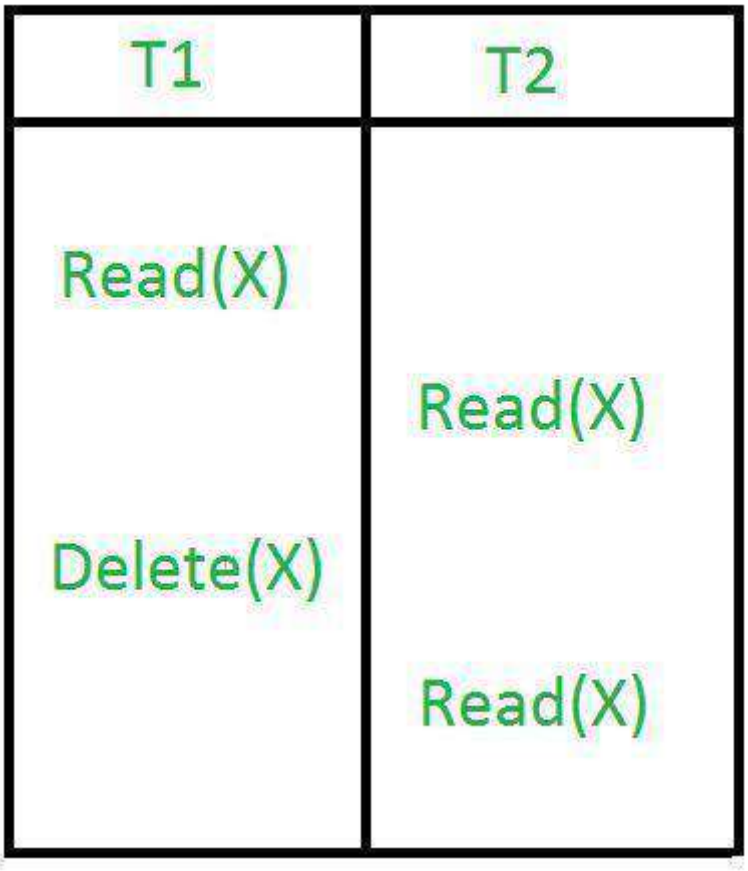


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

Phantom Read Problem:

The phantom read problem occurs **when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.**

Example:

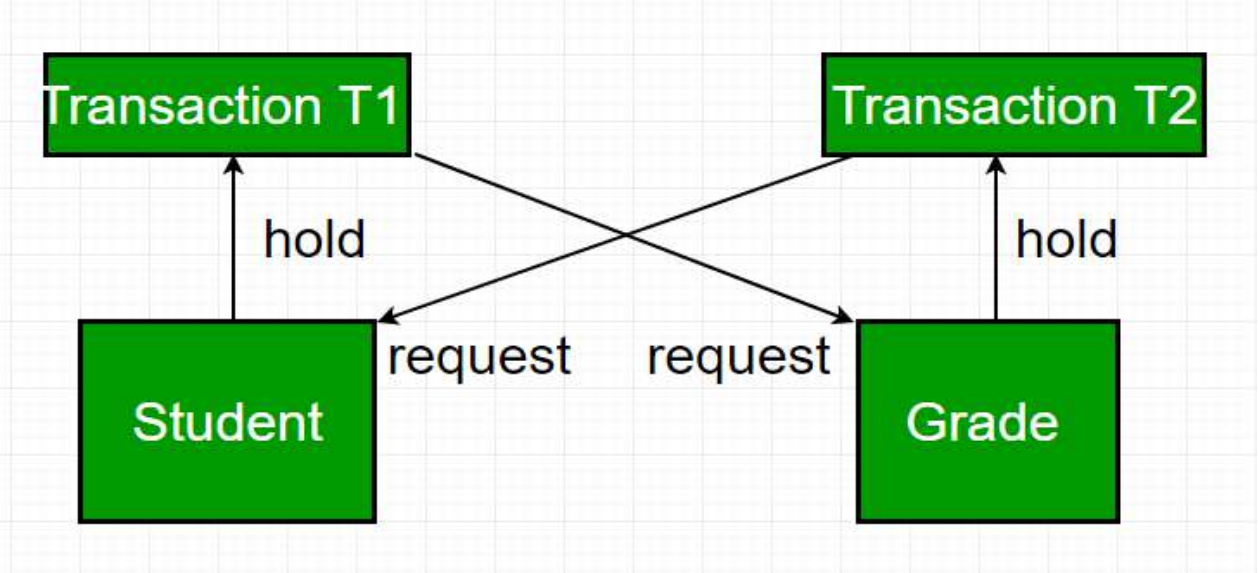


In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Deadlock in DBMS

A deadlock occurs in a **multi-user database environment when two or more transactions block each other indefinitely by each holding a resource the other needs**. This results in a cycle of dependencies (circular wait) where no transaction can proceed.

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.

How to Handle Deadlocks

There are some approaches and by ensuring them, we can handle deadlocks. They are discussed below:

1. Deadlock Avoidance

Plan transactions in a way that prevents deadlock from occurring.
Methods:

- Access resources in the same order. For e.g., always access Students first, then Grades

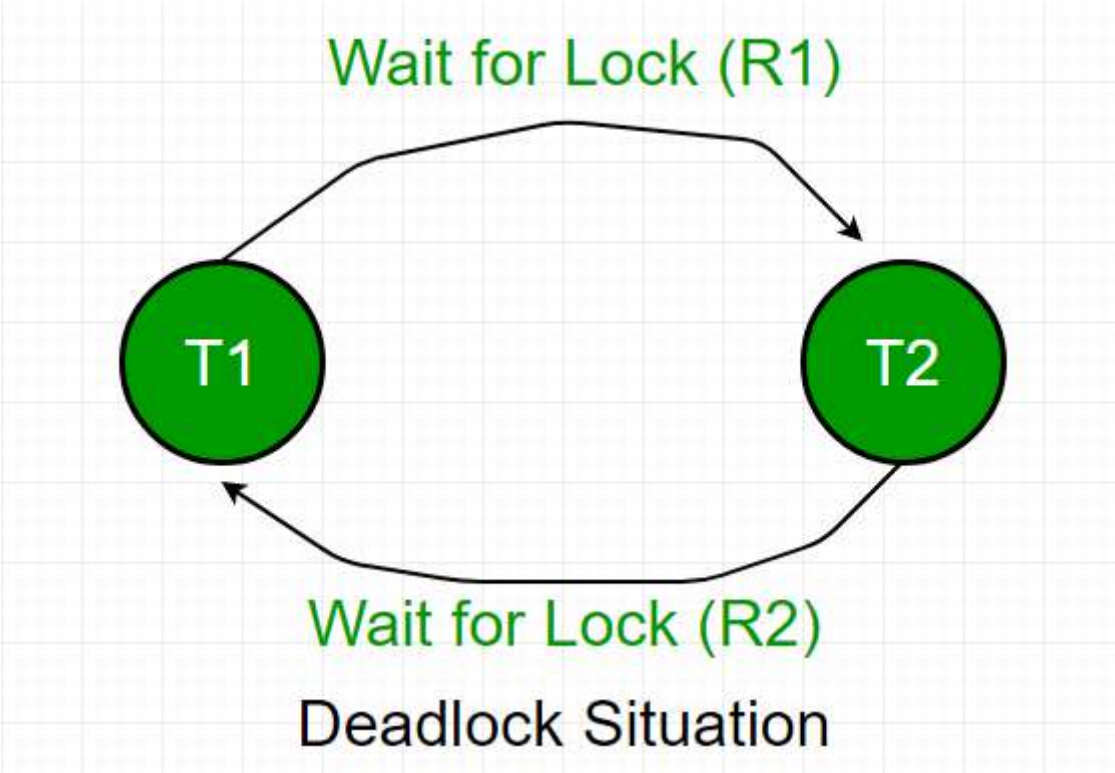
- Use row-level locking and READ COMMITTED isolation level. It reduces chances, but doesn't eliminate deadlocks.completely

2. Deadlock Detection

If a transaction waits too long, the DBMS checks if it's part of a deadlock.

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



3. Deadlock Prevention

For a large database, the deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that a deadlock never occurs. The DBMS analyzes the operations whether they can create a deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes:

1. Wait-Die Scheme (Non-preemptive)

- **Older transactions are allowed to wait**.
- **Younger transactions are killed** (aborted and restarted) **if they request a resource held by an older one**

For example:

- Consider two transaction- T1 = 10 and T2 = 20
- If T1 (older) wants a resource held by T2 → T1 waits
- If T2 (younger) wants a resource held by T1 → T2 dies and restarts

Prevents deadlock by not allowing a younger transaction to wait and form a wait cycle.

2. Wound-Wait Scheme (Preemptive)

- Older transactions are aggressive (preemptive) and can **force younger ones to abort.**
- **Younger transactions must wait** if they want a resource held by an older one.

For example:

- Consider two transaction- T1 = 10 and T2 = 20
- If T1 (older) wants a resource held by T2 → T2 is killed, T1 proceeds.
- If T2 (younger) wants a resource held by T1 → T2 waits

Prevents deadlock by not allowing younger transactions to block older ones.

The following table lists the differences between Wait-Die and Wound-Wait scheme prevention schemes:

Wait - Die	Wound -Wait
It is based on a non-preemptive technique.	It is based on a preemptive technique.
In this, older transactions must wait for the younger one to release its data items.	In this, older transactions never wait for younger transactions.
The number of aborts and rollbacks is higher in these techniques.	In this, the number of aborts and rollback is lesser.

Lock Based Concurrency Control Protocol in DBMS

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

1. **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.
2. **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.

Concurrency Control Protocols

Concurrency Control Protocols are the methods used to manage multiple transactions happening at the same time. They ensure that transactions are executed safely without interfering with each other, maintaining the accuracy and consistency of the database.

These protocols prevent issues like data conflicts, lost updates or inconsistent data by controlling how transactions access and modify data.

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 locks are held until the transaction commits or aborts.**

How Strict 2PL Works

Strict 2PL has two phases:

1. Growing Phase

A transaction can acquire locks (shared or exclusive).

A transaction **cannot release any lock** during this phase.

2. Shrinking Phase

After the **transaction releases its first lock, it cannot acquire any new lock.**

Strict 2PL adds one more rule:
→ **Exclusive (write) locks are held until the transaction commits or aborts.**

Shared locks may be released earlier depending on the implementation, **but exclusive locks must stay until commit.**

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**.

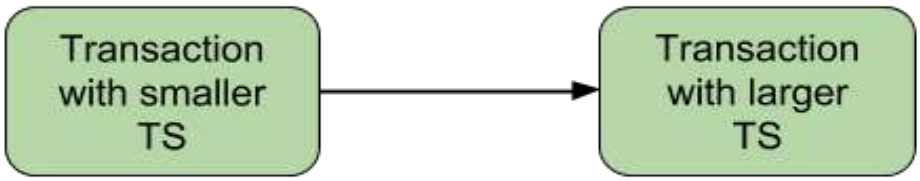
3. Ensuring Serializability: The protocol ensures that the schedule of transactions is serializable. This means the transactions **can be executed in an order** that is logically equivalent to their timestamp order.

How Timestamp Ordering Works

Each **data item X** in the database **keeps two timestamps:**
- W_TS(X): **Timestamp of the last transaction that wrote to X**
- R_TS(X): **Timestamp of the last transaction that read from X**

Basic Timestamp Ordering

The Basic TO Protocol **works by comparing the timestamp of the current transaction** with the **timestamps on the data items it wants to read/write:**



Precedence Graph for TS ordering

- 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 $TS(T_i) < TS(T_j)$.
- 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.

Two Basic TO protocols are discussed below:

1 When T issues R_item(X) (a read request)
Rules

1. **If W_TS(X) > TS(T) → Abort T**
    - This means: *a newer transaction (with a larger timestamp) already wrote X.*
    - If T were allowed to read X, it would read a value written **from the future**, violating timestamp order.
    - Therefore, to preserve serial order, T must abort.
2. **Else → Allow the read**
   Update:
3. R_TS(X) = max(R_TS(X), TS(T))

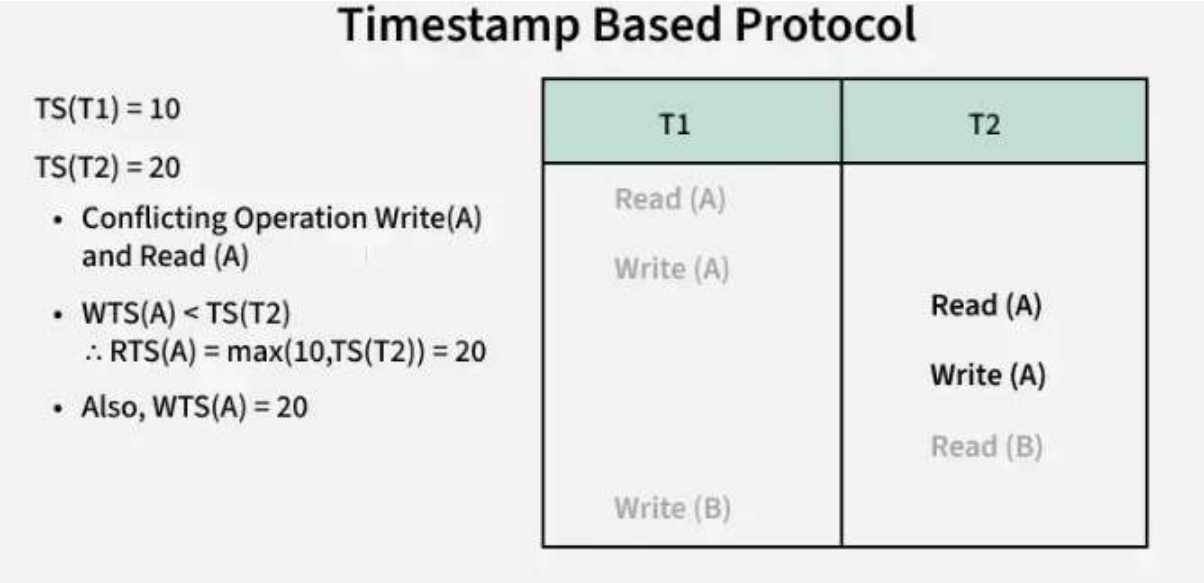
- If no future transaction has written X, T is allowed to read safely.
- We **update R_TS(X) to record that the most recent reader is now T (if T is newer than previous readers).**

2 When T issues W_item(X) (a write request)
Rules

1. **If R_TS(X) > TS(T) → Abort T**

    - Meaning: some *newer* transaction with timestamp greater than T has already read X.
    - Allowing T to write now would create a situation where a newer transaction read an "old" value, violating timestamp order.
    - To maintain order, T must abort.
2. **If W_TS(X) > TS(T) → Abort T**
    - Meaning: a newer transaction has already written X.
    - If we let T write now, it would produce data *from the past*, which contradicts the required timestamp order.
    - So T must abort.
3. **Else → Allow the write**
   Update:
4. W_TS(X) = TS(T)

- You can write only if **no newer transaction** has read or written X.
- **When allowed, the write becomes the newest write by setting W_TS(X).**



Strict Timestamp Ordering Protocol

The Strict Timestamp Ordering Protocol is an **enhanced version that avoids cascading rollbacks by delaying operations until it's safe to execute them**.

Strict TO modifies Basic TO by:

- **Allowing operations to wait** instead of aborting immediately
- **Requiring that all conflicting transactions commit before the operation proceeds**

This prevents cascading aborts and ensures *strictness*.

Key Features

- **Strict Execution Order**: Transactions **must execute in the exact order of their timestamps**. Operations are delayed if executing them would violate the timestamp order, ensuring a strict schedule.
- **No Cascading Rollbacks**: To avoid cascading aborts, **a transaction must delay its operations until all conflicting operations of older transactions are either committed or aborted**.
- **Consistency and Serializability**: The protocol ensures conflict-serializable schedules **by following strict ordering rules based on transaction timestamps.**

1️⃣ **Rules for Read Operation**: R_item(X)

A transaction T may read X only if:

1. W_TS(X) ≤ TS(T)
   - The **most recent write on X must be from an older (or equal) transaction.**
   - Meaning:
     No younger transaction has already written X.
     If a **younger transaction has written X, T is "too old" to read this newer value and must wait.**
2. The transaction that last wrote X has committed
   - T cannot read an uncommitted value.
   - Ensures strictness and eliminates cascading aborts.

If these are not satisfied:

The read is delayed (blocked) until conditions become true.

2️⃣ Rules for Write Operation: W_item(X)

A transaction T may write X only if:

1. R_TS(X) ≤ TS(T)
   - No younger transaction has already read X.
   - Otherwise, T would be producing a value *older* than what a newer transaction read.
2. W_TS(X) ≤ TS(T)
   - No younger transaction has already written X.
   - Otherwise, T would overwrite a "future" value.
3. All previous readers/writers of X have committed
   - Ensures:
     - **All older reads of X are finalized** (committed)
     - **All older writes are committed** (no uncommitted overwrites)
     - **Prevents cascading aborts**

If these aren't satisfied:

➜ The write is delayed, not aborted.

Advantages	Disadvantages
Conflict-Serializable: Maintains a correct execution order	**Cascading Rollbacks** (in Basic TO protocol)
Deadlock-Free: No locks, so no circular waits	**Starvation**: Newer transactions may be delayed
Simple Conflict Resolution: Uses timestamps only	**High Overhead**: Constantly updating R_TS/W_TS
No Locking Needed: Avoids lock management complexity	**Lower Throughput** under high concurrency

Advantages	Disadvantages
Predictable Execution: Operations follow a known order	**Delayed Execution** in Strict TO for consistency