DATABASE TRANSACTION PROCESSING, CONCURRENCY CONTROL & RECOVERY SYSTEM

Transaction Processing

- A transaction is a logical unit of database processing that includes one or more database access operations such as insertion, deletion, modification and retrieval.
- Database operations that form a transaction can either be embedded within an application program or they can be specified interactively through a high level query language such as SQL, where it is delimited by statements of the form *begin transaction* and *end transaction*. All database operations between these two points are considered as one transaction.

  e.g.

  begin transaction

  …

  … One Transaction

  …

  end transaction

Transaction Processing Contd…

- A Single application program may contain more than one transaction it it contains several boundaries.
- Transactions access data using two operations-
  - Read (X) – It transfers the data items X from the database to a local buffer belonging to the transaction that executed the read operation.
  - Write(X) – It transfers the data item X from the local buffer of the transaction that executed the write back of the database.
- Example of two very simple transactions.

T1	T2

read (X);
X := X-N;
write(X);
read (Y);
Y := Y+N;
write(Y);
	read (X);
	X := X+M;
	write(X);

Transaction T1	Transaction T2

Transaction Properties

- **The database system maintains the main four properties of the transactions to ensure integrity of the data.**

  i. **Atomicity – A Transaction is an atomic unit of processing. It is either performed in its entirely or not performed at all.**

  ii. **Consistency – Transaction preserve database consistency. That is, a transaction transform a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points.**

  iii. **Isolation – A Transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.**

  iv. **Durability or Permanency – The changes applied to the database by a committed transaction must persist in the database. These changes must not lost because of any failure.**

These properties are known as the ACID properties.

Transaction Properties

For Example: Consider a banking system consisting of several accounts and a set of Transactions that access and update those accounts to understand and need of ACID properties.

Let Ti be a transaction that transfer amount Rs. 500 from account A to account B.

This transaction can be defined as –

Ti:

```
read (A);
A := A-500;
write (A);
read (B);
B := B+500;
write (B);
```

Now, each of the ACID requirement is considered as follows-

Transaction Properties

1. *Consistency:* The consistency requirement here is that the sum of A & B be unchanged by the execution of the transaction. Without the consistency requirement, money can be created or destroyed by the transaction.

   Ensuring consistency for an individual transaction is the responsibility of the application programmer. This task may be facilitated by automatic testing of integrity constraints.

2. *Atomicity:* Suppose that, just before the execution of transaction Ti the values of A and B are 1000 and 2000. Now, suppose that the execution of transaction Ti, a failure happened after the write(A) operation but before the write (B) operation due to power failures, hardware failures and software failures. In this case the values of A and B reflected in the database are 950 and 2000. These system destroyed 50 as a result of this failure. In this case, the sum A+B is no longer preserved.

   Thus, the system reaches to an inconsistent state. We must ensure that such inconsistency are not visible in database system. However, this state is eventually replaced by the consistent state, where the value of account A is 950 and the value of account B is 2050. That is the reason for the atomicity requirement- if the atomicity property is present, all actions of transaction are reflected in the database, or none are. (Transaction-Management Task)

Transaction Properties

3. ***Durability:*** Once the user has been notified that the transaction has completed (i.e. the transfer of 50 has taken place), the update to the database by the transaction must persist despite failures.

   Thus the durability property guarantee that once a transaction completes successfully, all the updates that carried out on the database persist, even if there is a system failure after the transaction completes execution.

   (Recovery-Management Task)

4. ***Isolation:*** If between states 3 & 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be loss then it should be).

   The problem concurrently executing transactions are avoided by executing transactions serially i. e. one after the other. However, executing multiple transactions concurrently has significant benefits. (Concurrency-Control Task)

Transaction States

A Transaction must be in one of the following states-

(i) **Active:** It's initial state; the transaction stays in this state while it is executing.

(ii) **Partially Committed:** After the final statement has been executed.

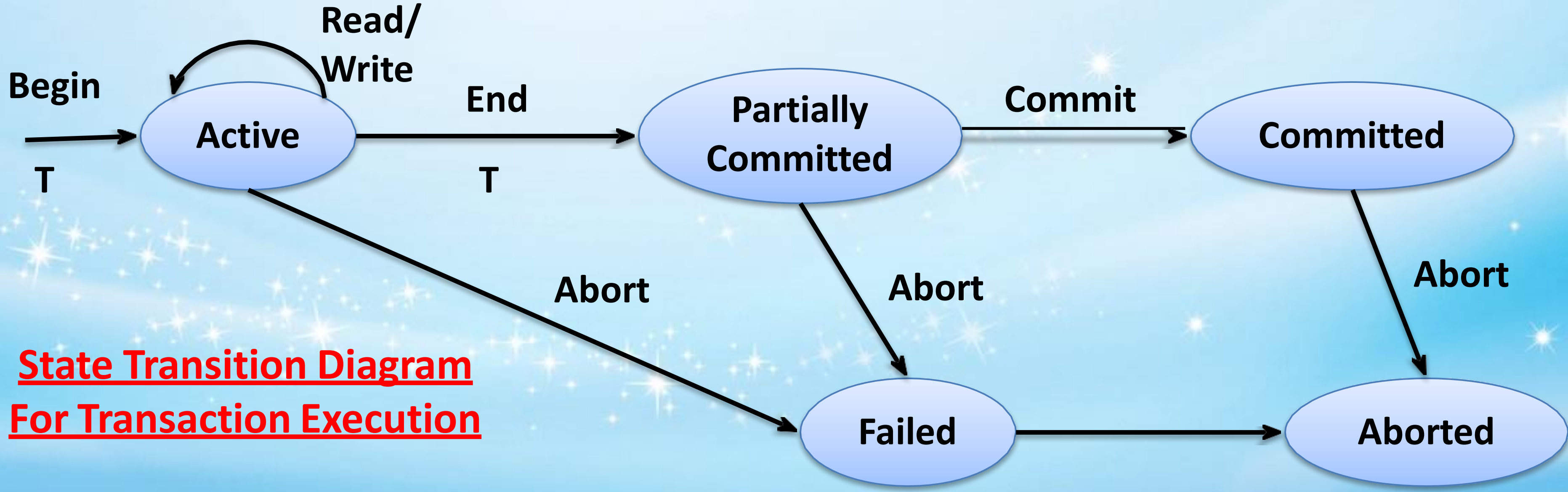
(iii) **Failed:** After the discovery that normal execution can no longer proceed.

(iv) **Aborted:** After the transaction has been called back and the database restored to its state prior to the start of the transaction. Two options are available after the transaction has been aborted:

a) Restart the transaction, only if there is no internal logical error.

b) Kill the transaction.

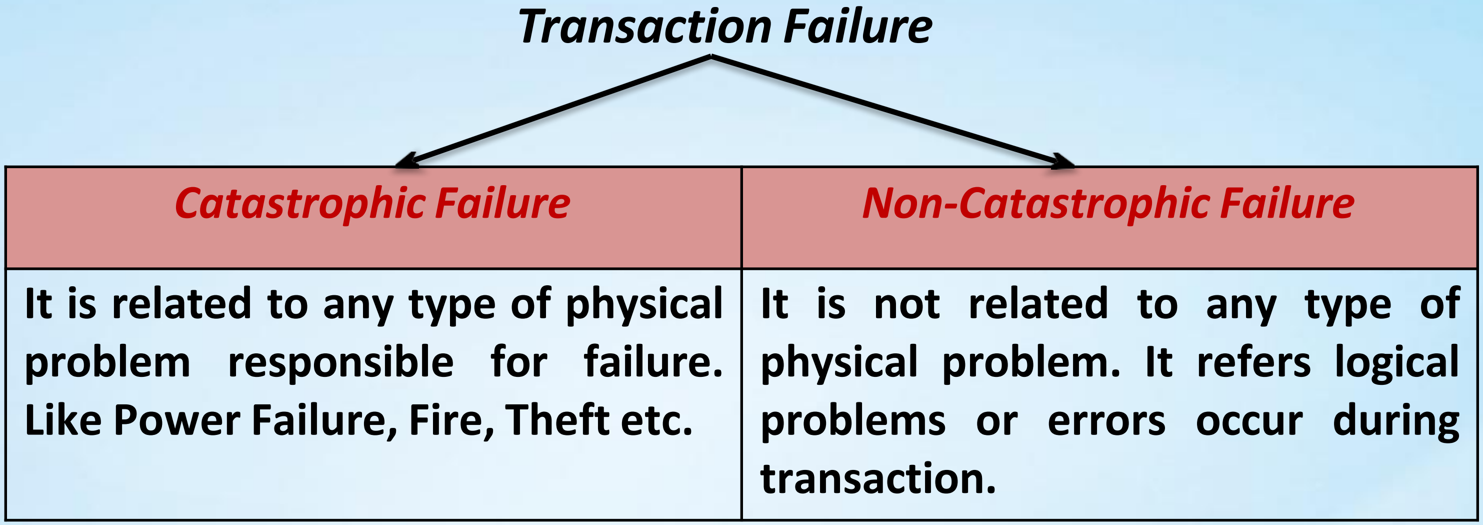
(v) **Committed:** After successful completion.



State Transition Diagram For Transaction Execution

Transaction Failure

A transaction may be failed due to many different reason during its execution time. A transaction can be aborted by several reasons from its any state of execution. There are mainly two different types of failure.



Transaction Failure

Catastrophic Failure	*Non-Catastrophic Failure*
It is related to any type of physical problem responsible for failure. Like Power Failure, Fire, Theft etc.	It is not related to any type of physical problem. It refers logical problems or errors occur during transaction.

There are many problems for transaction failure:

1. *System Crash: It refers to the hardware, software or network error occurs in the system during execution. Hardware crashes are usually media failure like Hard sick, RAM, Processor etc.*
2. *System Error: System errors are the errors which are occurred during execution of transaction. These errors are division by zero, integer overflow, buffer overflow etc. These errors occurs due to logical programming errors.*

Transaction Failure…

3. ***Exception or errors in transaction:*** *Some errors are occurred during execution of transaction which can not be handled by transactions known as Exception. These exceptions can be programmed in the transaction to overcome the failure. The exceptions are:*

   a) Data not found for transaction

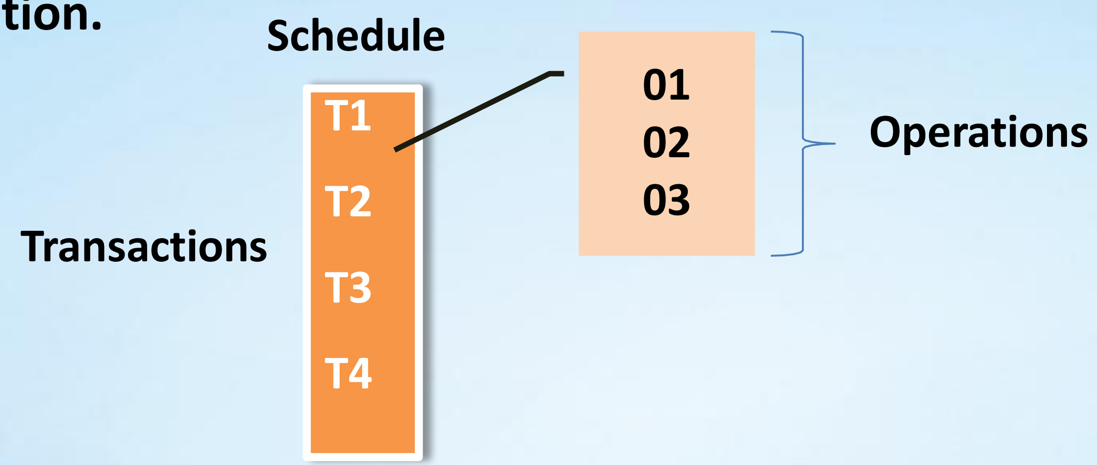
   b) Insufficient Account Balance

   c) Illegal Operation Performed

4. ***Catastrophic Failure or Physical Problems:*** *Physical Problems are endless list of problems like power failure, fire, theft, natural calamity, overwriting disk etc. Such problems do not happen frequently; if they occur, recovery is a major tack.*

Transaction Schedule

Schedule- A Schedule is an ordering sequence of the operations of the transactions. A schedule should follow the constraint that the sequence of operations for each transactions should be same as they appear in the transaction.



Schedule

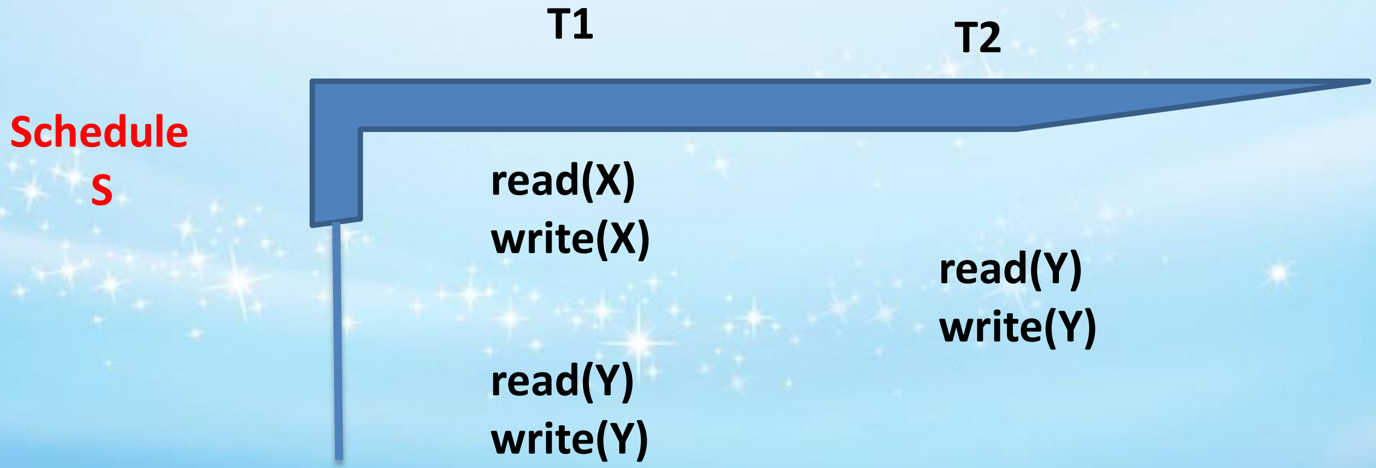
Transactions

T1
T2
T3
T4

01
02
03

Operations

In a schedule, sequence of operations of the transaction should not be changed if two transactions are interleaved in the process of execution.



Schedule S

T1	T2

read(X)
write(X)	

	read(Y)
	write(Y)

read(Y)
write(Y)	

Complete Schedule

A Schedule is n transactions T1, T2, T3 ……… Tn is said to be a complete schedule if the following conditions exists-

i) The operations in S are exactly those operations in T1, T2, T3………Tn, including a commit or abort operationas the last operation of each transaction in the schedule.

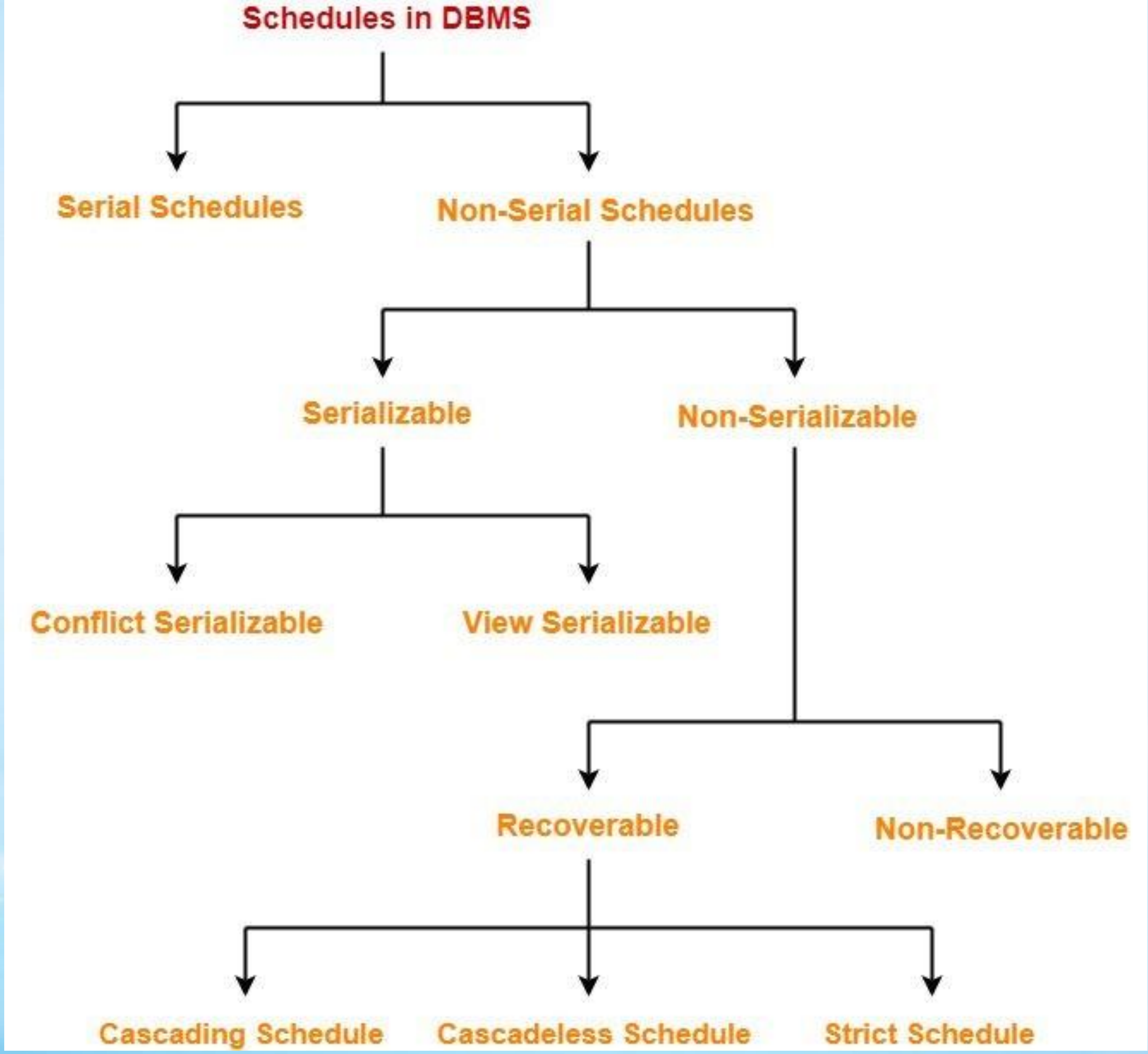
ii) For any pair of operations from the same transaction Ti, their order of appearance in S is the same as their order of appearance in Ti.

iii) For any two conflicting operations, one of the two must occur before the other in the schedule.

$$D = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ W(X) & & \\ Com. & & \\ & R(Y) & \\ & W(Y) & \\ & Com. & \\ & & R(Z) \\ & & W(Z) \\ & & Com. \end{bmatrix}$$

Complete Schedule

The order in which the operations of multiple transactions appear for execution is called as a schedule.



Schedules in DBMS

Serial Schedules

Non-Serial Schedules

Serializable

Non-Serializable

Conflict Serializable

View Serializable

Recoverable

Non-Recoverable

Cascading Schedule

Cascadeless Schedule

Strict Schedule

Serial Schedule

A Schedule is called Serial, if the operations of each transaction are executed consecutively without any interleaved operations from other transaction.

T1	T2

read (X);
X := X-N;
write(X);
read (Y);
Y := Y+N;
write(Y);	

	read (X);
	X := X+M;
	write(X);

Serial Schedule

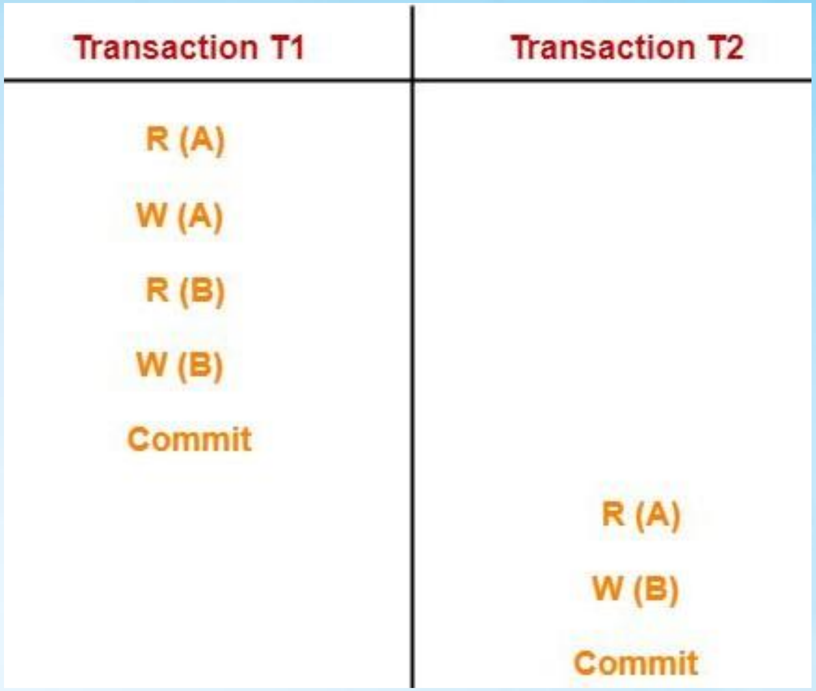
Time

A Schedule S is Serial, if for every transaction T participating in the schedule, all the operations of T are executed consecutively. Otherwise the schedule is called non-serial. Hence in a serial schedule, only one transaction at a time is active and no interleaving occurs.

Serial Schedule

Serial schedules are always-

- **Consistent**
- **Recoverable**
- **Cascadeless**
- **Strict**



Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

In this schedule,

1. There are two transactions T1 and T2 executing serially one after the other.
2. Transaction T1 executes first.
3. After T1 completes its execution, transaction T2 executes.

So, this schedule is an example of a Serial Schedule.

Non-serial Schedule

A Schedule is called Non-serial, if each sequence interleaves operations from two transactions.

T1	T2

read (X);
X := X-N;
write(X);	

	read (X);
	X := X+M;
	write(X);

read(Y);
Y := Y+N;
write(Y);	

Non-Serial Schedule

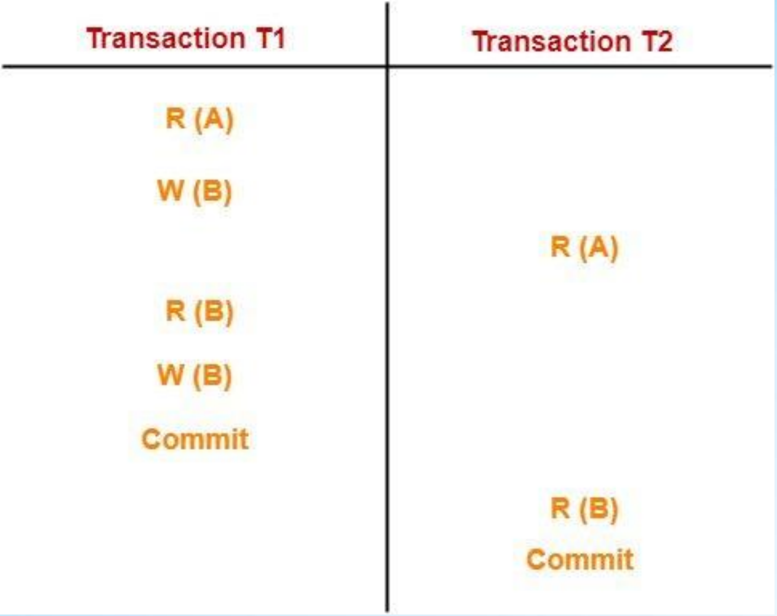
Time

Interleaving of transactions T1 and T2

Non-serial schedules are NOT always-

- Consistent
- Recoverable
- Cascadeless
- Strict

Non-serial Schedule



Transaction T1	Transaction T2
R (A)	
W (B)	
	R (A)
R (B)	
W (B)	
Commit	
	R (B)
	Commit

In this schedule,

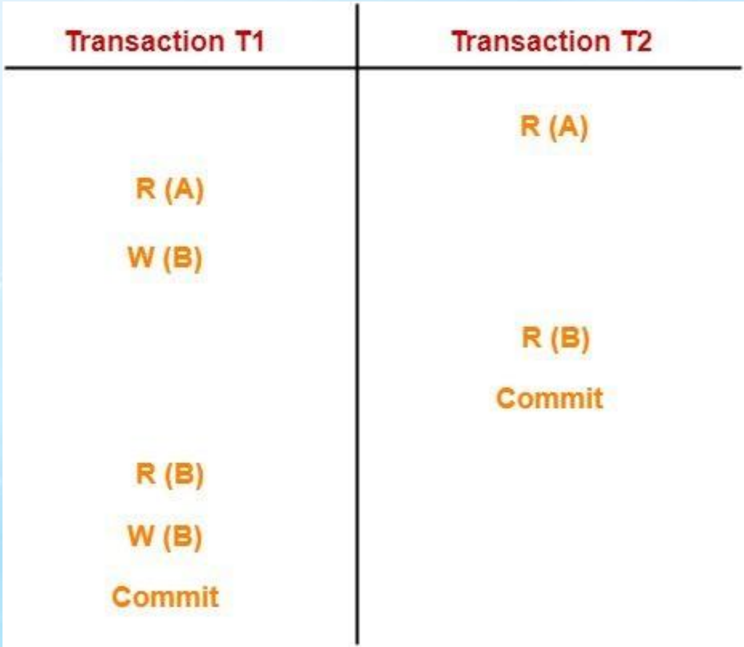
- There are two transactions T1 and T2 executing concurrently.
- The operations of T1 and T2 are interleaved.

So, this schedule is an example of a **Non-Serial Schedule**.

In this schedule,

- There are two transactions T1 and T2 executing concurrently.

•The operations of T1 and T2 are interleaved. So, this schedule is an example of a **Non-Serial Schedule.**



Transaction T1	Transaction T2
	R (A)
R (A)	
W (B)	
	R (B)
	Commit
R (B)	
W (B)	
Commit	

Finding Number of Schedules

Finding Number Of Schedules

Consider there are n number of transactions T1, T2, T3 …. , Tn with N1, N2, N3 …. , Nn number of operations respectively.

Total Number of Schedules

Total number of possible schedules (serial + non-serial) is given by-

$$\frac{(N1 + N2 + N3 + ... + Nn)!}{N1! \times N2! \times N3! \times .... \times Nn!}$$

Total Number of Serial Schedules

Total number of serial schedules =

Number of different ways of arranging n transactions = n!

Total Number of Non-Serial Schedules

Total number of non-serial schedules =

Total number of schedules – Total number of serial schedules

Finding Number of Schedules

Problem

Consider there are three transactions with 2, 3, 4 operations respectively, find-

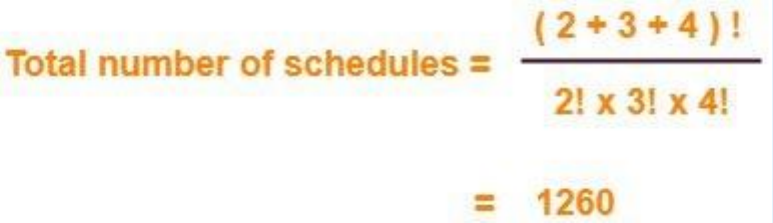
How many total number of schedules are possible?

How many total number of serial schedules are possible?

How many total number of non-serial schedules are possible?

Total Number of Schedules

Using the above formula, we have



$$\text{Total number of schedules} = \frac{(2+3+4)!}{2! \times 3! \times 4!}$$

$$= 1260$$

Total Number of Serial Schedules

Total number of serial schedules

= Number of different ways of arranging 3 transactions

= 3!

= 6

Total Number of Non-Serial Schedules

Total number of non-serial schedules

= Total number of schedules – Total number of serial schedules

= 1260 – 6

= 1254

Serializability

The concept of Serializability of schedules is used to identify which schedules are correct when the transaction executions have interleaving of their operations in the schedules.

A Schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. Each serial schedule consists of a sequence of instructions from the various transactions, where the instructions belonging to one single transaction appear together in that schedule.

A non-serial schedule S is serializable is equivalent to say that it is correct schedule, because it is same as a serial schedule.

Serializability

Serializability

- Some non-serial schedules may lead to inconsistency of the database.
- Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

Serializable Schedules

If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a **serializable schedule**.

Serializable schedules behave exactly same as serial schedules.

Thus, serializable schedules are always-

- Consistent
- Recoverable
- Cascadeless
- Strict

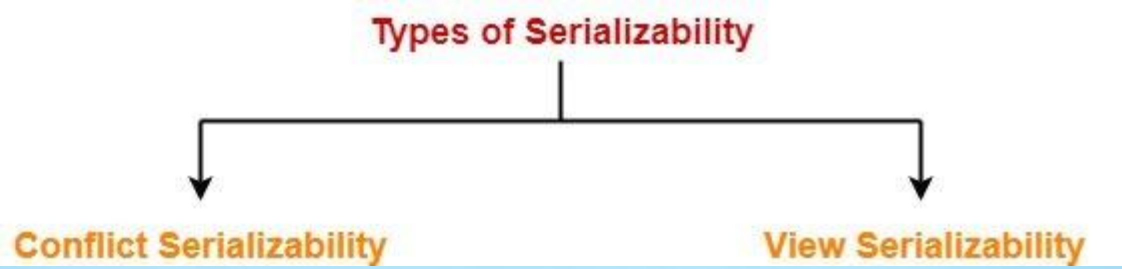
Serial Schedules Vs Serializable Schedules

Serial Schedules	Serializable Schedules
No concurrency is allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
Serial schedules lead to less resource utilization and CPU throughput.	Serializable schedules improve both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.

Types of Serializability

Serializability is mainly of two types-

1. Conflict Serializability
2. View Serializability



Conflict Serializability

Conflict Serializability-

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Conflicting Operations-

Two operations are called as **conflicting operations** if all the following conditions hold true for them-

1. Both the operations belong to different transactions
2. Both the operations are on the same data item
3. At least one of the two operations is a write operation

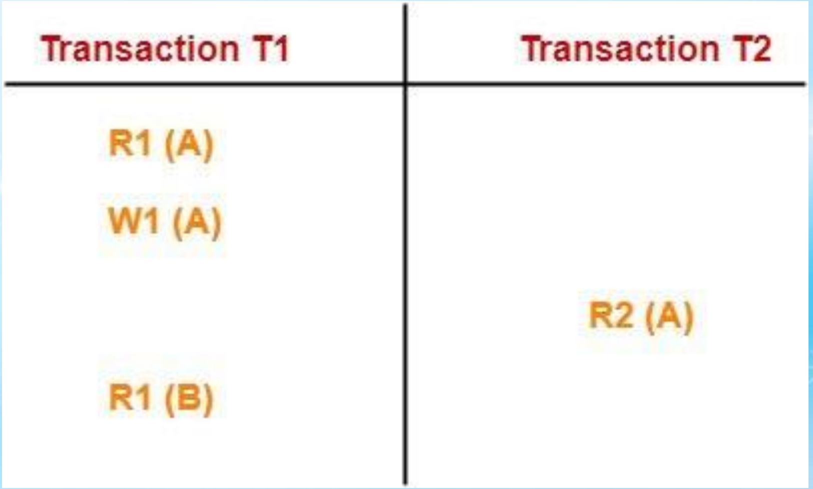
Example-

Consider the following schedule-

In this schedule,

- W1 (A) and R2 (A) are called as conflicting operations.

This is because all the above conditions hold true for them.



Transaction T1	Transaction T2
R1 (A)	
W1 (A)	
	R2 (A)
R1 (B)	

Conflict Serializability

Follow the following steps to check whether a given non-serial schedule is conflict serializable or not-

Step-01:

Find and list all the conflicting operations.

Step-02:

Start creating a precedence graph by drawing one node for each transaction.

Step-03:

Draw an edge for each conflict pair such that if X_i (V) and Y_j (V) forms a conflict pair then draw an edge from T_i to T_j.

This ensures that T_i gets executed before T_j.

Step-04:

Check if there is any cycle formed in the graph.

If there is no cycle found, then the schedule is conflict serializable otherwise not.

Conflict Serializability

Problem-01:

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

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

T1	T2	T3
R1 (A)		
	R2 (A)	
R1 (B)		
	R2 (B)	
		R3 (B)
W1(A)		
	W2(B)	

Solution

Step-01:

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

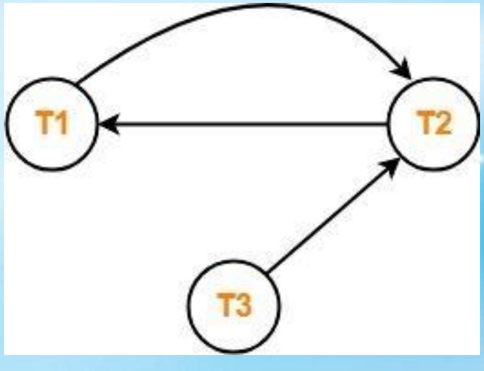
$R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$

$R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$

$R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Step-02:

Draw the precedence graph-

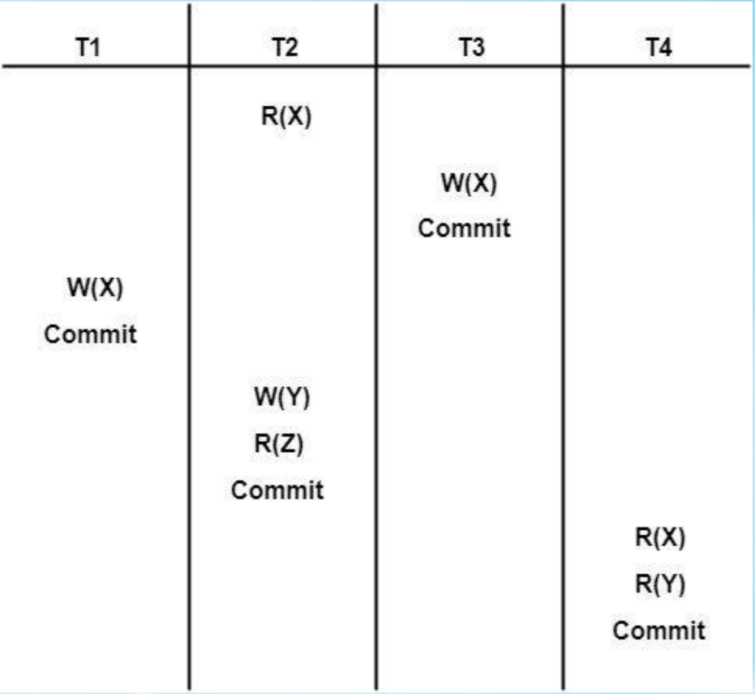


- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

Conflict Serializability

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

Checking Whether S is Conflict Serializable Or Not-

Step-01:

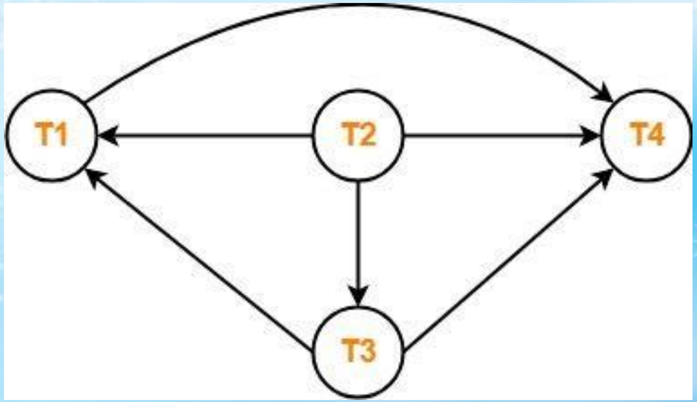
List all the conflicting operations and determine the depe

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

Step-02:

Draw the precedence graph-

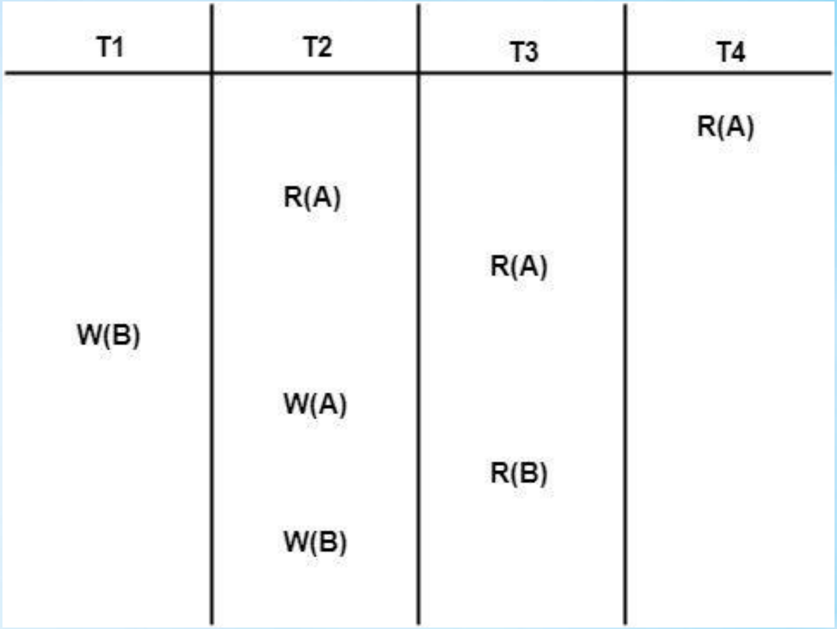
- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.



Conflict Serializability

Problem-03:

Check whether the given schedule S is conflict serializable or not. If yes, then determine all the possible serialized schedules-



T1	T2	T3	T4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

Checking Whether S is Conflict Serializable Or Not

Step-01:

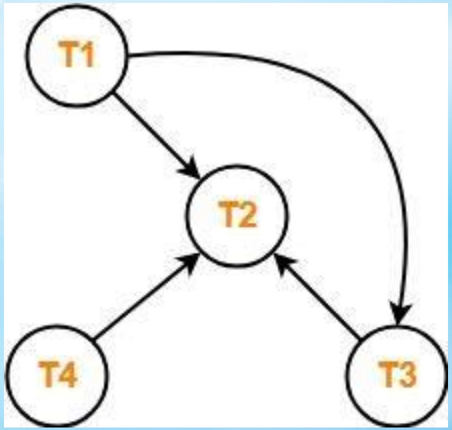
List all the conflicting operations and determine the dependency between the transactions-

$R_4(A)$, $W_2(A)$ $(T_4 \rightarrow T_2)$
$R_3(A)$, $W_2(A)$ $(T_3 \rightarrow T_2)$
$W_1(B)$, $R_3(B)$ $(T_1 \rightarrow T_3)$
$W_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
$R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Step-02:

Draw the precedence graph-

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



View Serializability

If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view serializable schedule.

View Equivalent Schedules

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2.

Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them:

Condition-01

For each data item X, if transaction T_i reads X from the database initially in schedule S1, then in schedule S2 also, T_i must perform the initial read of X from the database.

Thumb Rule: “Initial readers must be same for all the data items”.

Condition-02

If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S1, then in schedule S2 also, transaction T_i must read the same data item that has been updated by the transaction $T_{j.}$

Thumb Rule: “Write-read sequence must be same.”.

Condition-03

For each data item X, if X has been updated at last by transaction T_i in schedule S1, then in schedule S2 also, X must be updated at last by transaction T_i.

Thumb Rule: “Final writers must be same for all the data items”.

View Serializability

Checking Whether a Schedule is View Serializable Or Not

Method-01

Check whether the given schedule is conflict serializable or not.

1. If the given schedule is conflict serializable, then it is surely view serializable. Stop and report your answer.
2. If the given schedule is not conflict serializable, then it may or may not be view serializable. Go and check using other methods.

Rules

- All conflict serializable schedules are view serializable.
- All view serializable schedules may or may not be conflict serializable.

Method-02:

Check if there exists any blind write operation.

(Writing without reading is called as a blind write).

1. If there does not exist any blind write, then the schedule is surely not view serializable. Stop and report your answer.
2. If there exists any blind write, then the schedule may or may not be view serializable. Go and check using other methods.

Rules

- No blind write means not a view serializable schedule.

View Serializability

Method-03:

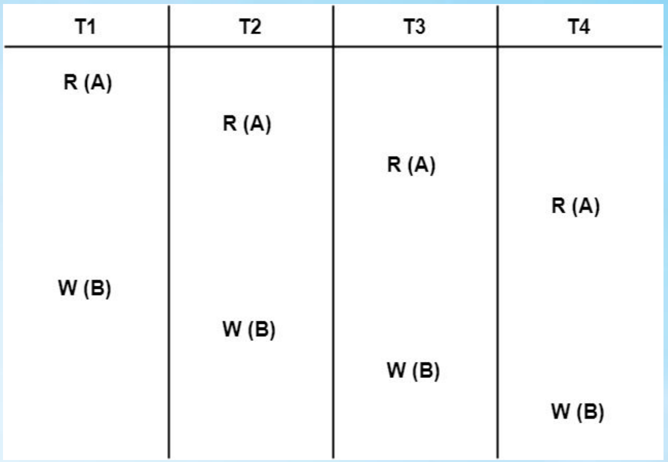
In this method, try finding a view equivalent serial schedule.

1. By using the above three conditions, write all the dependencies.
2. Then, draw a graph using those dependencies.
3. If there exists no cycle in the graph, then the schedule is view serializable otherwise not.

View Serializability

Problem-01

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



T1	T2	T3	T4
R (A)			
	R (A)		
		R (A)	
			R (A)
W (B)			
	W (B)		
		W (B)	
			W (B)

Solution-

We know, if a schedule is conflict serializable, then it is surely view serializable.

So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not

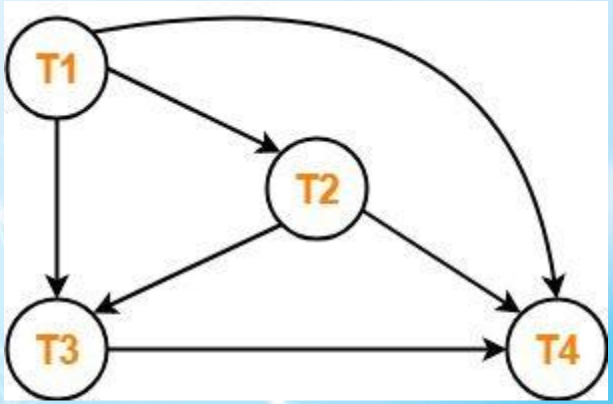
Step-01:

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

$W_1(B)$, $W_2(B)$	$(T_1 \rightarrow T_2)$
$W_1(B)$, $W_3(B)$	$(T_1 \rightarrow T_3)$
$W_1(B)$, $W_4(B)$	$(T_1 \rightarrow T_4)$
$W_2(B)$, $W_3(B)$	$(T_2 \rightarrow T_3)$
$W_2(B)$, $W_4(B)$	$(T_2 \rightarrow T_4)$
$W_3(B)$, $W_4(B)$	$(T_3 \rightarrow T_4)$

Step-02:

Draw the precedence graph-



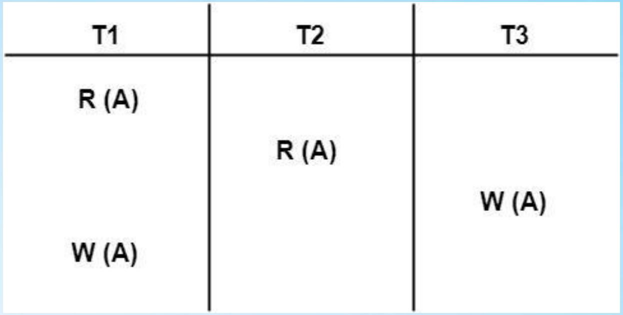
- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.

Thus, we conclude that the given schedule is also view serializable.

View Serializability

Problem-02:

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



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

Solution-

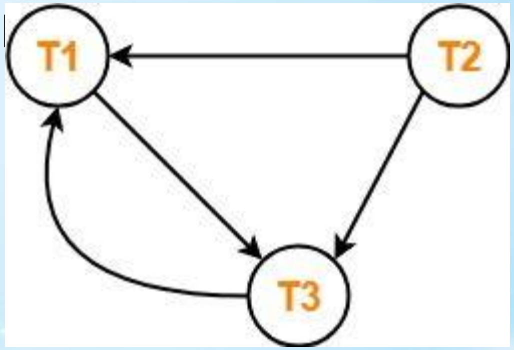
We know, if a schedule is conflict serializable, then it is surely view serializable. So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

Step-01:

List all the conflicting operations and determine the dependency

$R_1(A)$, $W_3(A)$	$(T_1 \rightarrow T_3)$
$R_2(A)$, $W_3(A)$	$(T_2 \rightarrow T_3)$
$R_2(A)$, $W_1(A)$	$(T_2 \rightarrow T_1)$
$W_3(A)$, $W_1(A)$	$(T_3 \rightarrow T_1)$



Step-02:

Draw the precedence graph-

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

View Serializability

Now,

- Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable. To check whether S is view serializable or not, let us use another method.

Let us check for blind writes.

Checking for Blind Writes

- There exists a blind write $W_3(A)$ in the given schedule S.
- Therefore, the given schedule S may or may not be view serializable.

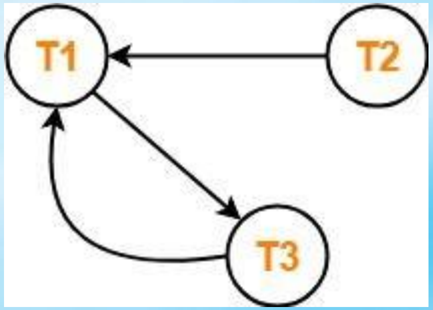
Now,

To check whether S is view serializable or not, let us use another method.

Let us derive the dependencies and then draw a dependency graph.

Drawing a Dependency Graph-

- T1 firstly reads A and T3 firstly updates A.
- So, T1 must execute before T3.
- Thus, we get the dependency **T1 → T3**.
- Final updation on A is made by the transaction T1.
- So, T1 must execute after all other transactions.
- Thus, we get the dependency **(T2, T3) → T1**.
- There exists no write-read sequence.



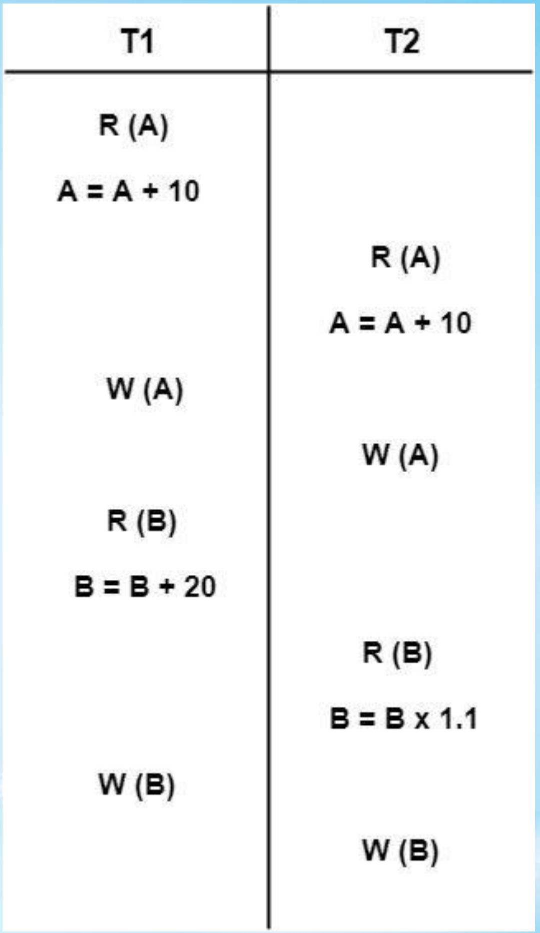
Now, let us draw a dependency graph using these dependencies-

- Clearly, there exists a cycle in the dependency graph.
- Thus, we conclude that the given schedule S is not view serializable.

View Serializability

Problem-03:

Check whether the given schedule S is view serializable or not



T1	T2
R (A)	
A = A + 10	
	R (A)
	A = A + 10
W (A)	
	W (A)
R (B)	
B = B + 20	
	R (B)
	B = B x 1.1
W (B)	
	W (B)

Solution-

We know, if a schedule is conflict serializable, then it is surely view serializable. So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

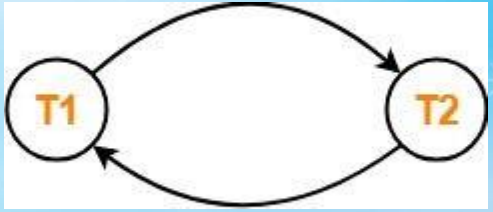
Step-01:

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

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

Step-02:

Draw the precedence graph-



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

View Serializability

Now,

- Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable. To check whether S is view serializable or not, let us use another method.

Let us check for blind writes.

Checking for Blind Writes-

- There exists no blind write in the given schedule S.
- Therefore, it is surely not view serializable.

Alternatively,

- You could directly declare that the given schedule S is not view serializable.
- This is because there exists no blind write in the schedule.
- You need not check for conflict Serializability.

Non-Serializable Schedule

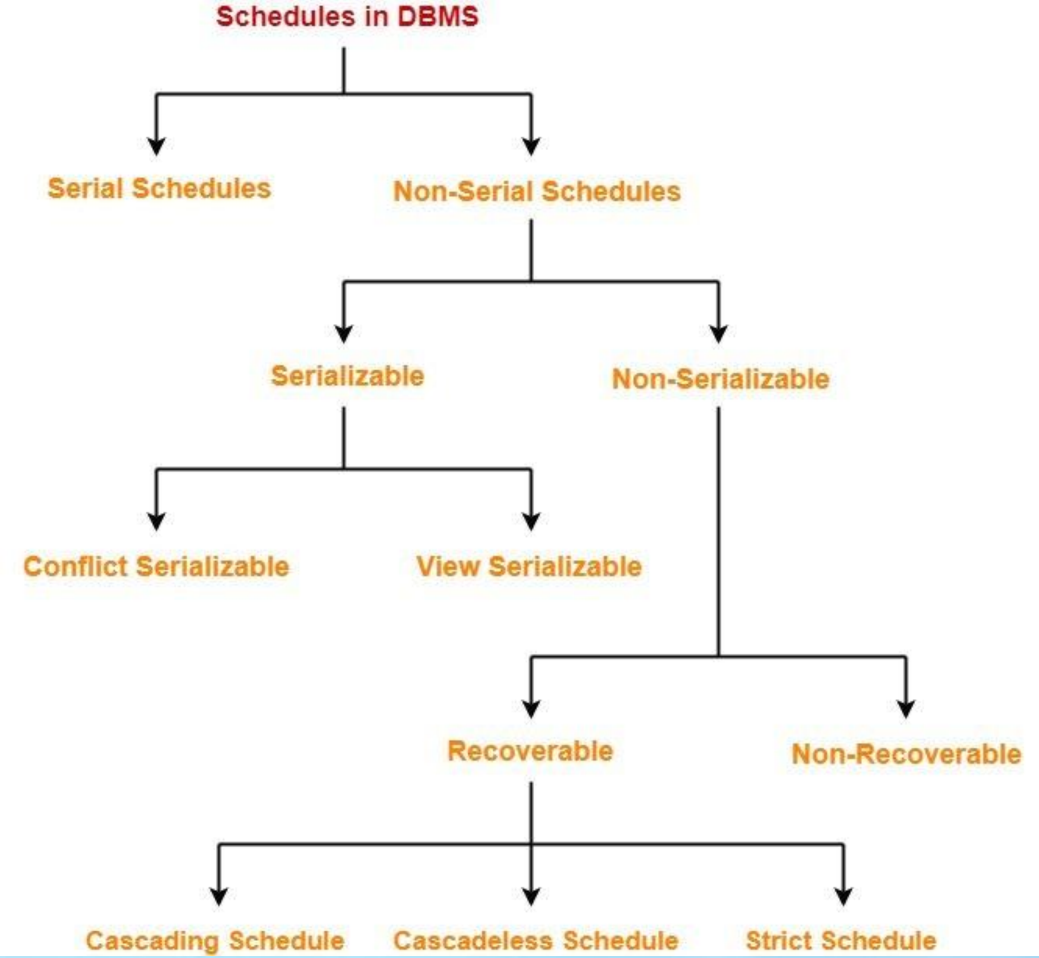
Non-Serializable Schedules-

- A non-serial schedule which is not serializable is called as a non-serializable schedule.
- A non-serializable schedule is not guaranteed to produce the same effect as produced by some serial schedule on any consistent database.

Characteristics-

Non-serializable schedules

- may or may not be consistent
- may or may not be recoverable



Non-Serializable Schedule

Irrecoverable Schedules (Non-recoverable)

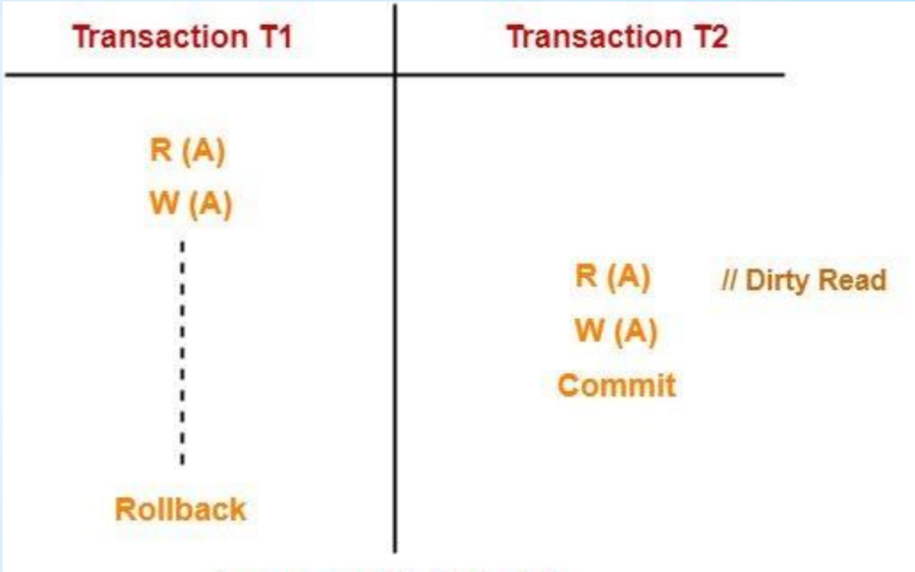
If in a schedule,

- A transaction performs a dirty read operation from an uncommitted transaction
- And commits before the transaction from which it has read the value

then such a schedule is known as an **Irrecoverable Schedule**.

Example-

Consider the given schedule-



Transaction T1	Transaction T2	
R (A)		
W (A)		
	R (A)	// Dirty Read
	W (A)	
	Commit	
Rollback		

Irrecoverable Schedule

Here,

- T2 performs a dirty read operation.
- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 can not recover since it has already committed.

Non-Serializable Schedule

Recoverable Schedules

If in a schedule,

- A transaction performs a dirty read operation from an uncommitted transaction
- And its commit operation is delayed till the uncommitted transaction either commits or roll backs

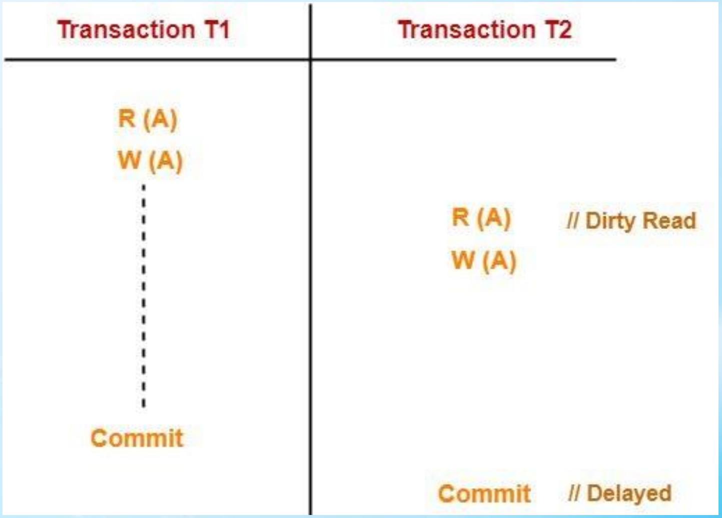
then such a schedule is known as a **Recoverable Schedule**.

Here,

- The commit operation of the transaction that performs the dirty read is delayed.
- This ensures that it still has a chance to recover if the uncommitted transaction fails later.

Example-

Consider the following schedule-



Transaction T1	Transaction T2
R (A)	
W (A)	
	R (A) // Dirty Read
	W (A)
Commit	
	Commit // Delayed

Here,

- T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.

Non-Serializable Schedule

Checking Whether a Schedule is Recoverable or Irrecoverable-

Method-01:

Check whether the given schedule is conflict serializable or not.

- If the given schedule is conflict serializable, then it is surely recoverable. Stop and report your answer.
- If the given schedule is not conflict serializable, then it may or may not be recoverable. Go and check using other methods.

Rules

- All conflict serializable schedules are recoverable.
- All recoverable schedules may or may not be conflict serializable.

Non-Serializable Schedule

Method-02:

Check if there exists any dirty read operation.

(Reading from an uncommitted transaction is called as a dirty read)

- If there does not exist any dirty read operation, then the schedule is surely recoverable. Stop and report your answer.
- If there exists any dirty read operation, then the schedule may or may not be recoverable.

If there exists a dirty read operation, then follow the following cases-

Case-01:

If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then the schedule is irrecoverable.

Case-02:

If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

Rule

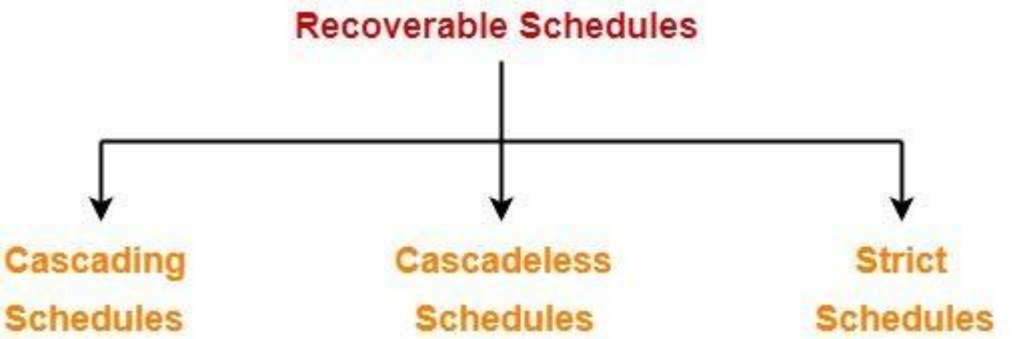
No dirty read means a recoverable schedule.

Non-Serializable Schedule

Types of Recoverable Schedules

A recoverable schedule may be any one of these kinds-

1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule



Cascading Schedule

- If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a **Cascading Schedule** or **Cascading Rollback** or **Cascading Abort**.
- It simply leads to the wastage of CPU time.

Non-Serializable Schedule

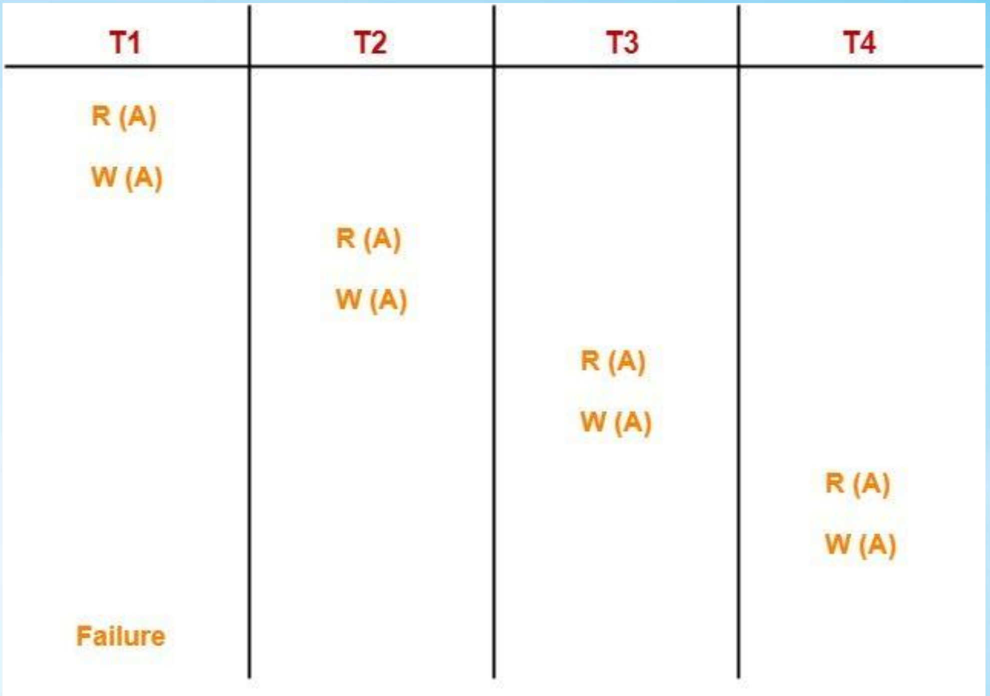
Example

Here,

Transaction T2 depends on transaction T1.

Transaction T3 depends on transaction T2.

Transaction T4 depends on transaction T3.



T1	T2	T3	T4
R (A)			
W (A)			
	R (A)		
	W (A)		
		R (A)	
		W (A)	
			R (A)
			W (A)
Failure			

Cascading Recoverable Schedule

In this schedule,

- The failure of transaction T1 causes the transaction T2 to rollback.
- The rollback of transaction T2 causes the transaction T3 to rollback.
- The rollback of transaction T3 causes the transaction T4 to rollback.

Such a rollback is called as a **Cascading Rollback**.

NOTE

If the transactions T2, T3 and T4 would have committed before the failure of transaction T1, then the schedule would have been irrecoverable.

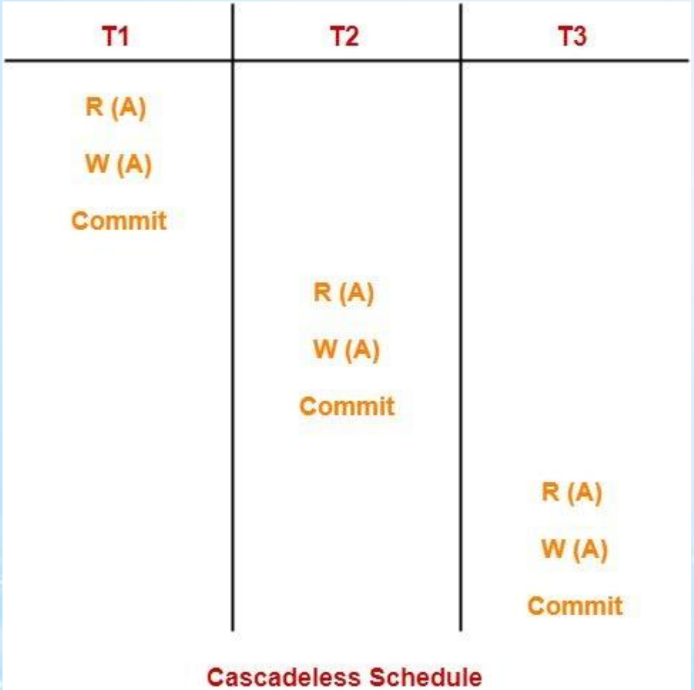
Non-Serializable Schedule

Cascadeless Schedule-

If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Cascadeless Schedule**.

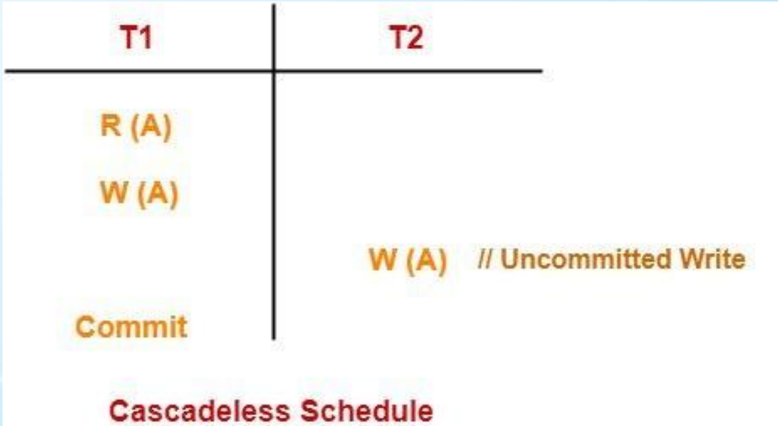
In other words,

- Cascadeless schedule allows only committed read operations.
- Therefore, it avoids cascading roll back and thus saves CPU time.



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

Cascadeless Schedule



T1	T2
R (A)	
W (A)	
	W (A) // Uncommitted Write
Commit	

Cascadeless Schedule

NOTE

- Cascadeless schedule allows only committed read operations.
- However, it allows uncommitted write operations.

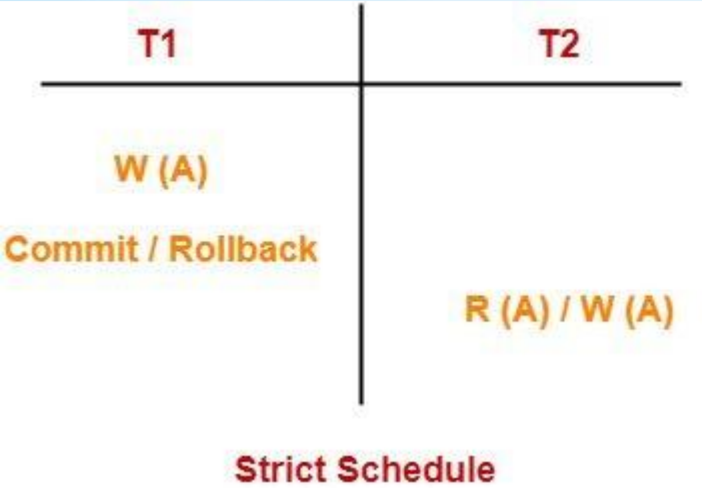
Non-Serializable Schedule

Strict Schedule

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.

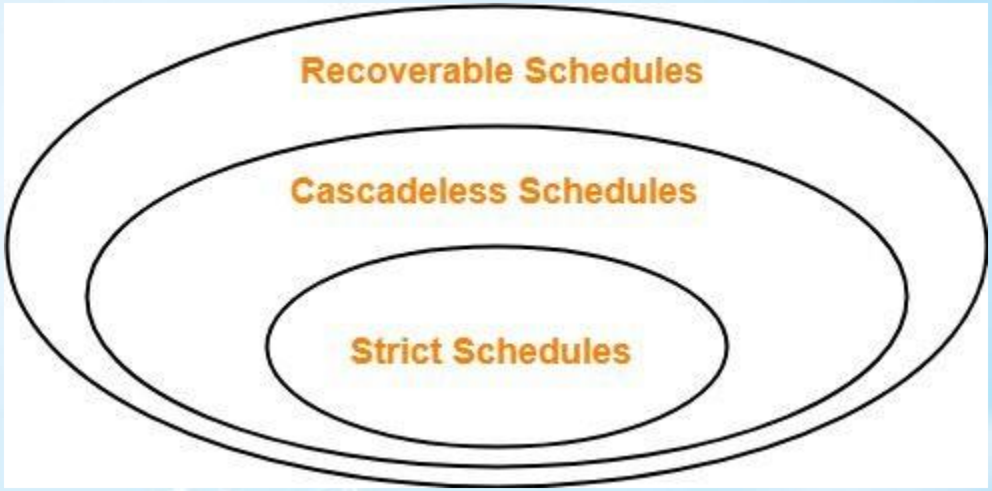
In other words,

- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than Cascadeless Schedule.



T1	T2
W (A)	
Commit / Rollback	
	R (A) / W (A)

Strict Schedule



Remember

- Strict schedules are more strict than Cascadeless schedules.
- All strict schedules are Cascadeless schedules.
- All Cascadeless schedules are not strict schedules.

Concurrency Control

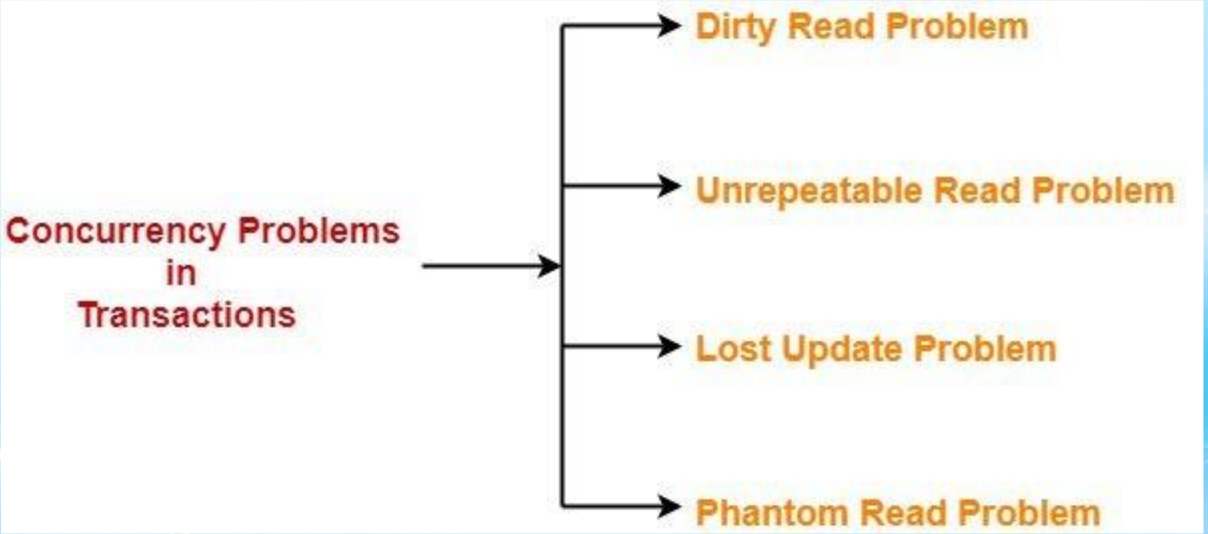
Concurrency Problems in DBMS

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.

Such problems are called as **concurrency problems**.

The concurrency problems are

1. Dirty Read Problem
2. Unrepeatable Read Problem
3. Lost Update Problem
4. Phantom Read Problem


Concurrency Problems in Transactions
Dirty Read Problem
Unrepeatable Read Problem
Lost Update Problem
Phantom Read Problem

Concurrency Control

Dirty Read Problem

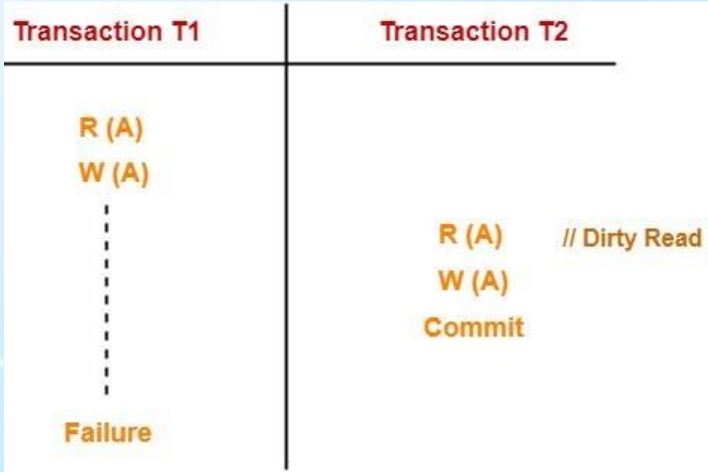
Reading the data written by an uncommitted transaction is called as dirty read.

This read is called as dirty read because-

- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.

NOTE-

- Dirty read does not lead to inconsistency always.
- It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.



Transaction T1	Transaction T2
R (A)	
W (A)	
	R (A) // Dirty Read
	W (A)
	Commit
Failure	

Here,

- T1 reads the value of A.
- T1 updates the value of A in the buffer.
- T2 reads the value of A from the buffer.
- T2 writes the updated the value of A.
- T2 commits.

- T1 fails in later stages and rolls back. In this example,

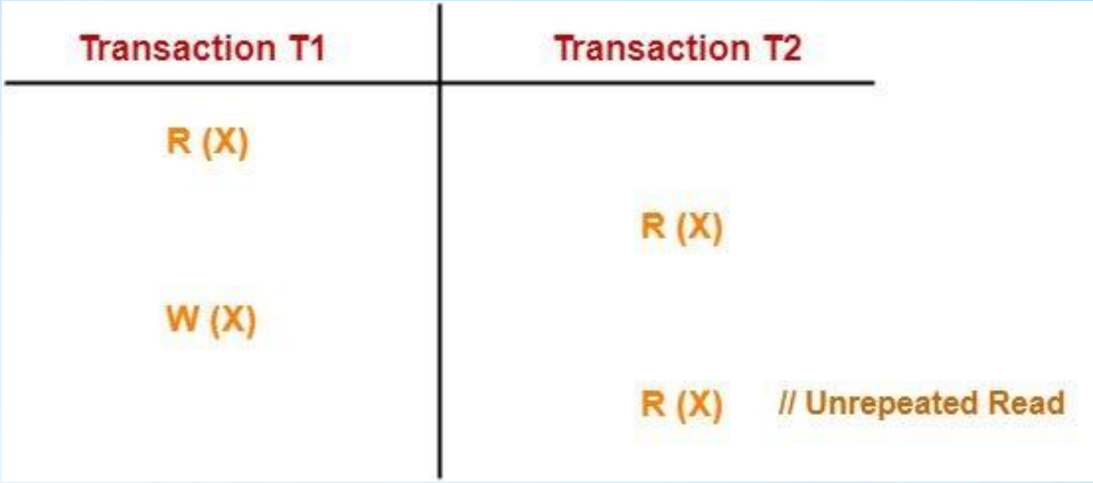
- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

Concurrency Control

Unrepeatable Read Problem

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Example



Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read

Here,

- T1 reads the value of X (= 10 say).
- T2 reads the value of X (= 10).
- T1 updates the value of X (from 10 to 15 say) in the buffer.
- T2 again reads the value of X (but = 15).

In this example,

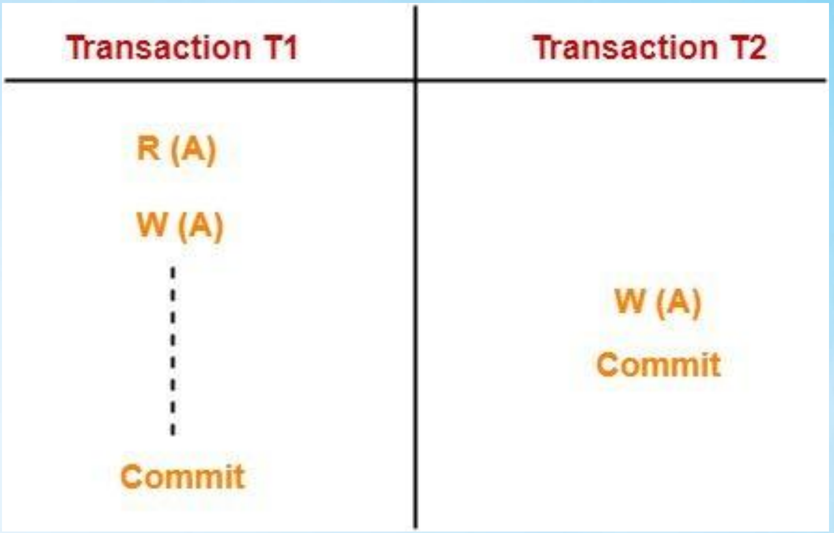
- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed because according to it, it is running in isolation.

Concurrency Control

Lost Update Problem

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.

Example



Transaction T1	Transaction T2
R (A)	
W (A)	
⋮	W (A)
⋮	Commit
Commit	

Here,

- T1 reads the value of A (= 10 say).
- T1 updates the value to A (= 15 say) in the buffer.
- T2 does blind write A = 25 (write without read) in the buffer.
- T2 commits.
- When T1 commits, it writes A = 25 in the database.

In this example,

- T1 writes the over written value of X in the database.
- Thus, update from T1 gets lost.

NOTE

This problem occurs whenever there is a write-write conflict.

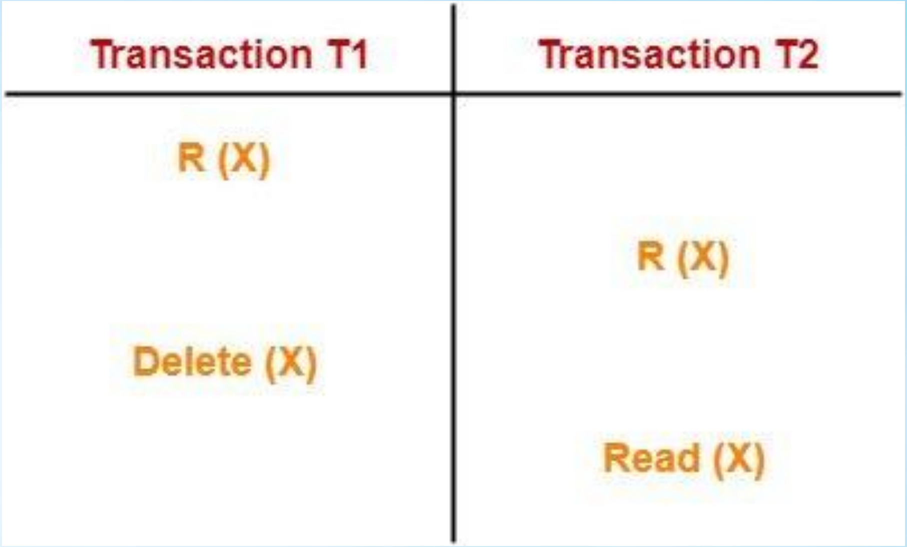
In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

Concurrency Control

Phantom Read Problem

This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.

Example



Transaction T1	Transaction T2
R (X)	
	R (X)
Delete (X)	
	Read (X)

Here,

- T1 reads X.
- T2 reads X.
- T1 deletes X.
- T2 tries reading X but does not find it.

In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation.

Avoiding Concurrency Problems

To ensure consistency of the database, it is very important to prevent the occurrence of above problems.

Concurrency Control Protocols help to prevent the occurrence of above problems and maintain the consistency of the database.

Concurrency Control

Avoiding Concurrency Problems

To ensure consistency of the database, it is very important to prevent the occurrence of above problems.

Concurrency Control Protocols help to prevent the occurrence of above problems and maintain the consistency of the database.

Concurrency control protocols ensure atomicity, isolation, and Serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1. Lock based protocol
2. Time-stamp protocol
3. Validation based protocol

Concurrency Control

Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it.

- A ***lock*** is a variable associated with a data item that describes the status of the data item with respect to the possible operations that can be applied to it. Manipulation the value of a lock is called ***locking***.

- Generally there is a one lock for each data item in the database.

- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Locks are of two kinds:

1. **Binary Locks** − A lock on a data item can be in two states; it is either locked or unlocked.
2. **Shared/exclusive** − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

Concurrency Control

Shared/Exclusive Lock − This type of locking mechanism differentiates the locks based on their uses. any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.
- If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but can not write Q.

2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.
- If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

Concurrency Control

A compatibility function on a set of lock modes can be defined as follows - Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj (Ti ≠ Tj) currently holds a lock of mode B. If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then mode A is compatible with mode B, This function can be represented by a matrix-

The compatibility relation between the two modes is shown in the matrix of figure. An element comp (A, B) of the matrix has the value true if and only if mode A is compatible with mode B.

- A shared mode is compatible with shared mode, but not with exclusive mode.
- At any time, several shared-mode locks can be held simultaneously by different transactions on a particular data item
- A subsequent exclusive mode lock request has to wait until the currently held shared mode lock are released.

	S	X
S	True	False
X	False	False

Concurrency Control

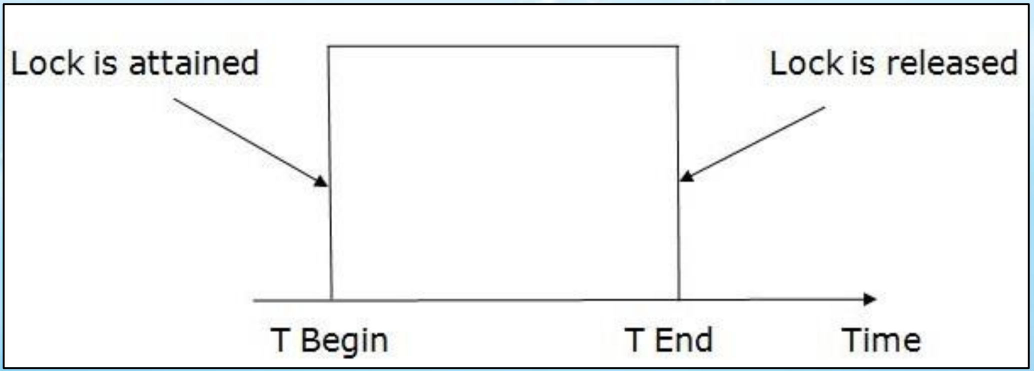
There are four types of lock protocols available:

1. Simplistic lock protocol

- It is the simplest way of locking the data while transaction.
- Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it.
- It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

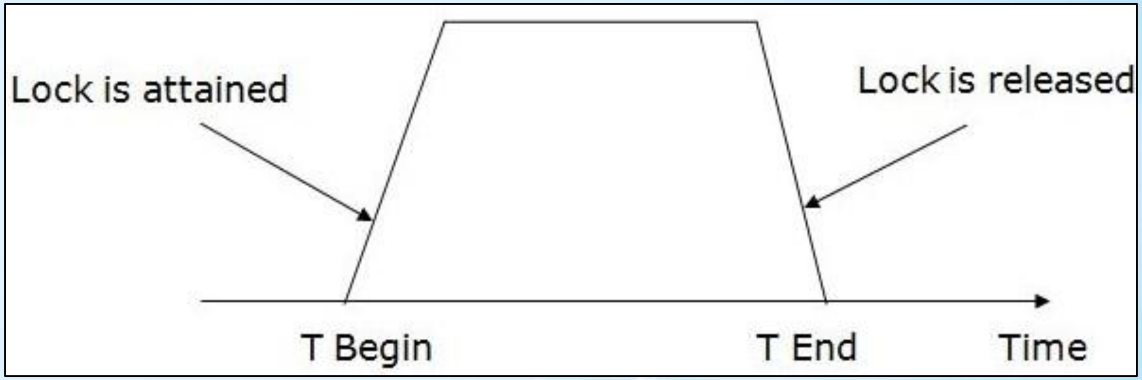
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



Concurrency Control

3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

2. **Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
3. **Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

- Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
- Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Concurrency Control

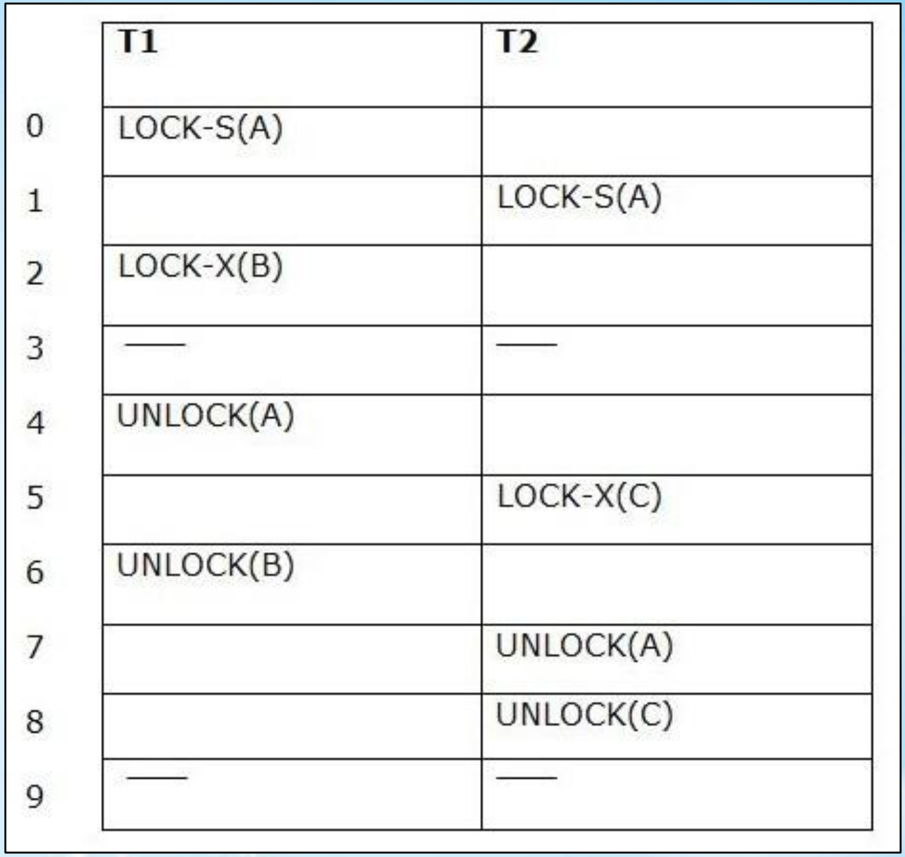
The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

Transaction T2:

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

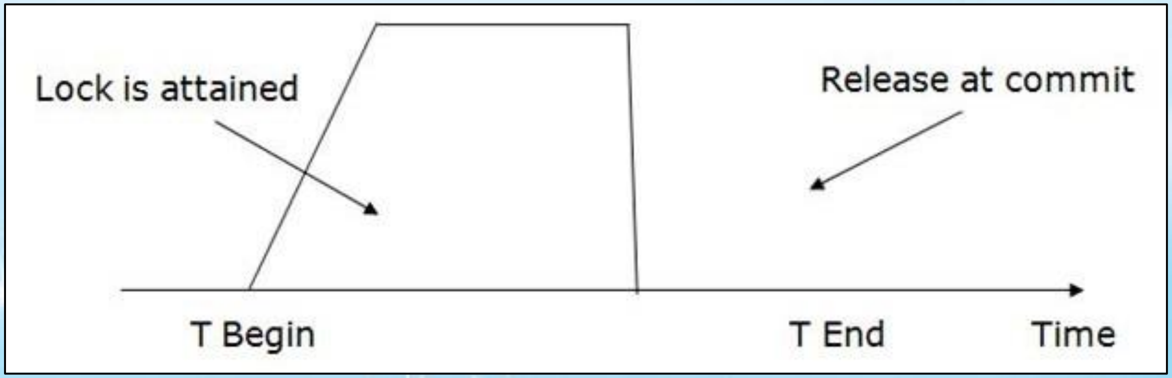


	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

Concurrency Control

4. **Strict Two-phase locking (Strict-2PL)**

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.
- It does not have cascading abort as 2PL does.



Concurrency Control

Timestamp Based Protocol

- Timestamp Based Protocol helps DBMS to identify the transactions.
- It is a unique identifier. Each transaction is issued a timestamp when it enters into the system.
- Timestamp protocol determines the Serializability order.
- It is most commonly used concurrency protocol.
- It uses either system time or logical counter as a timestamp.
- It starts working as soon as a transaction is created.

Timestamp Ordering Protocol

- The TO Protocol ensures Serializability among transactions in their conflicting read and write operations.

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.

- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

Concurrency Control

- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

- The transaction of timestamp (T) is denoted as TS(T).
- Data item (X) of read timestamp is denoted by R–TS(X).
- Data item (X) of write timestamp is denoted by W–TS(X).

There are three types of the Timestamp Ordering Protocol.

1. **Basic Timestamp Ordering Protocol**
2. **Validation Based Protocol**
3. **Thomas Write Rule**

Concurrency Control

1. **Basic Timestamp ordering protocol works as follows:**

Condition 1:

- Check the following condition whenever a transaction Ti issues a **Read (X)** operation:
  - If W_TS(X) >TS(Ti) then the operation is rejected.
  - If W_TS(X) <= TS(Ti) then the operation is executed.
- Timestamps of all the data items are updated.

Condition 2:

- Check the following condition whenever a transaction Ti issues a **Write(X)** operation:
  - If TS(Ti) < R_TS(X) then the operation is rejected.
  - If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Advantages and Disadvantages of TO protocol:

- TO protocol ensures Serializability since the precedence graph is as follows:

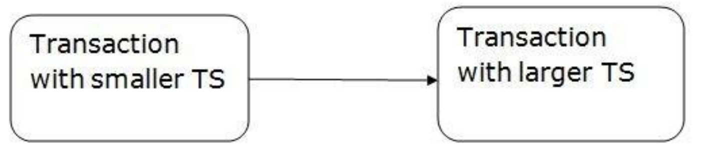


Image: Precedence Graph for TS ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade-free.

Concurrency Control

Revisited

Rule 1

If **TS(Ti) < W-TS(X),** this violates timestamp order of **Ti** with regard to the writer of **X**.

→ Abort **Ti** and restart it with same TS.

Else:

→ Allow **Ti** to read **X**.

→ Update **R-TS(X)** to **max(R-TS(X), TS(Ti)**)

→ Have to make a local copy of X to ensure repeatable reads for Ti .

Rule 2

If **TS(Ti) < R-TS(X)** or **TS(Ti) < W-TS(X)**

→ Abort and restart **Ti** .

Else:

→ Allow **Ti** to write **X** and update **W-TS(X)**

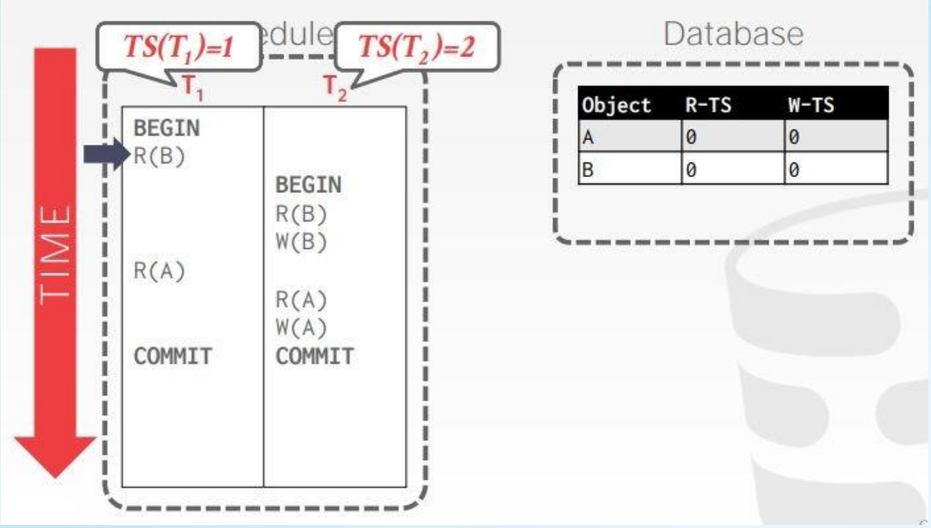
→ Also have to make a local copy of **X** to ensure repeatable reads for **Ti** .

Concurrency Control

Basic Timestamp ordering protocol

Example:

1



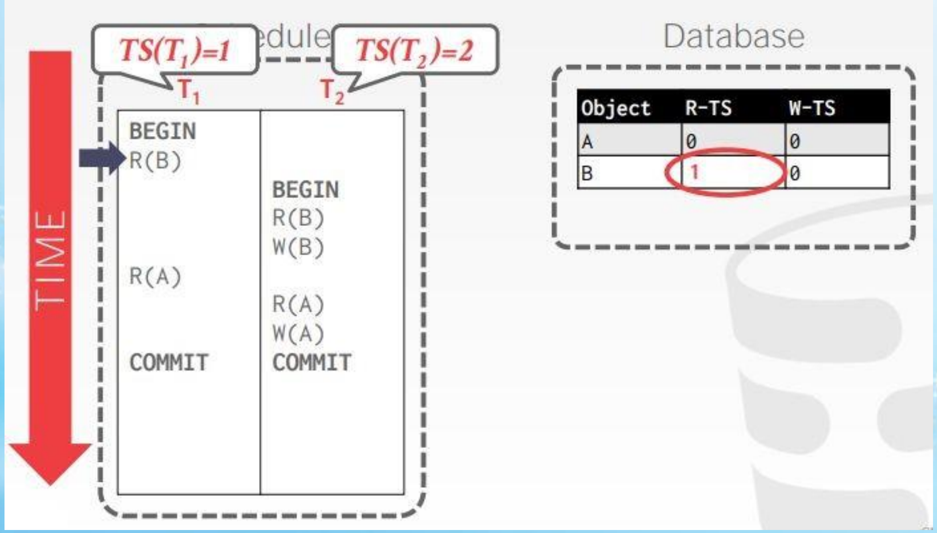
$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
→ R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	0	0

2



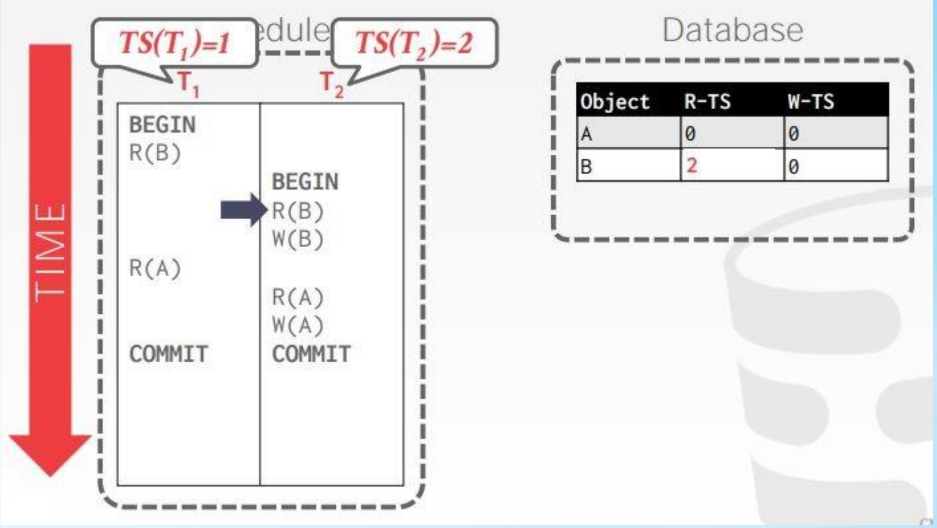
$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
→ R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	1	0

3



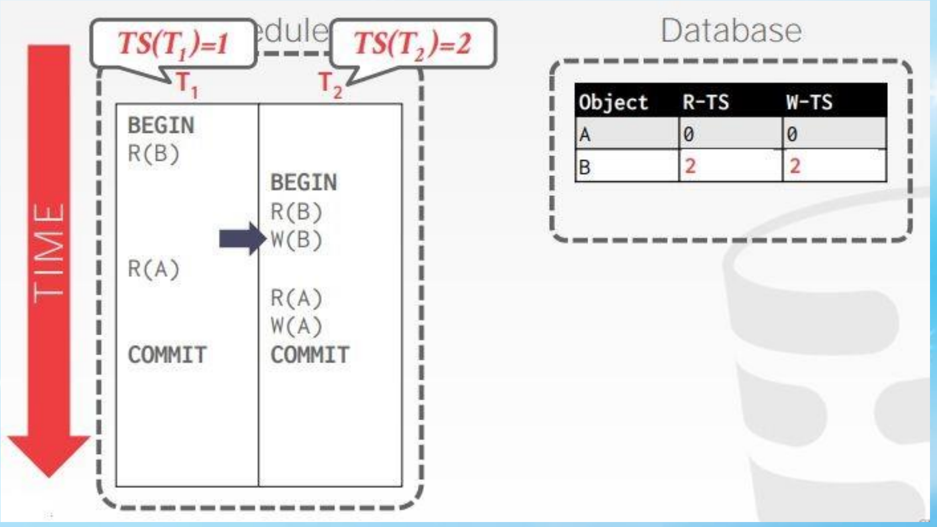
$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	→ R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	2	0

4



$TS(T_1)=1$ $TS(T_2)=2$

T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	→ W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

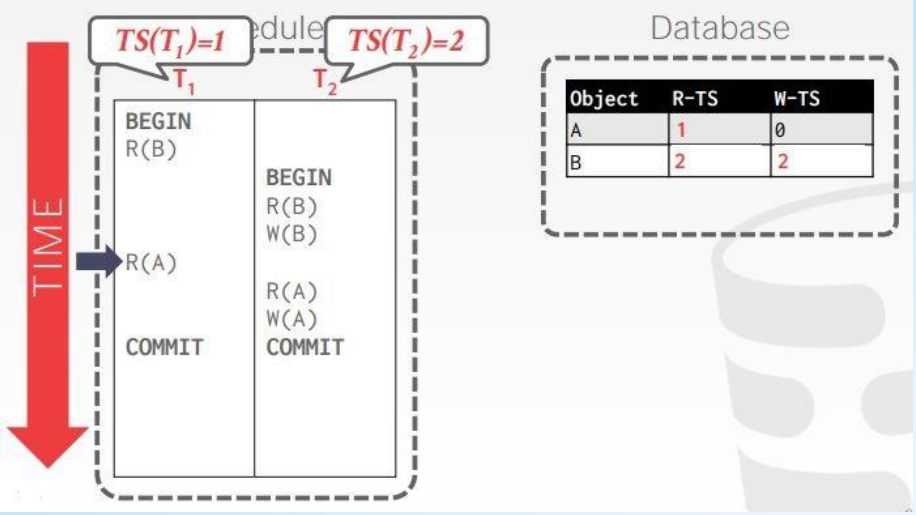
Database

Object	R-TS	W-TS
A	0	0
B	2	2

Basic Timestamp ordering protocol

Example:

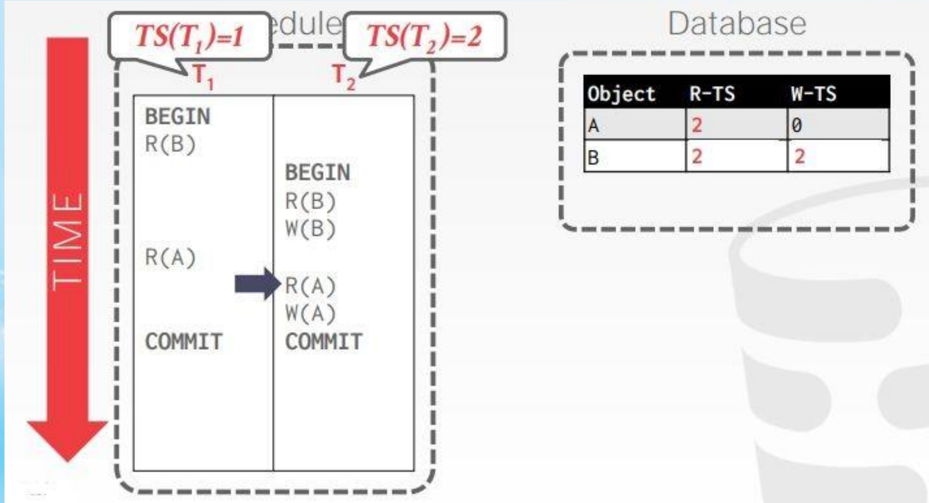
5



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

Object	R-TS	W-TS
A	1	0
B	2	2

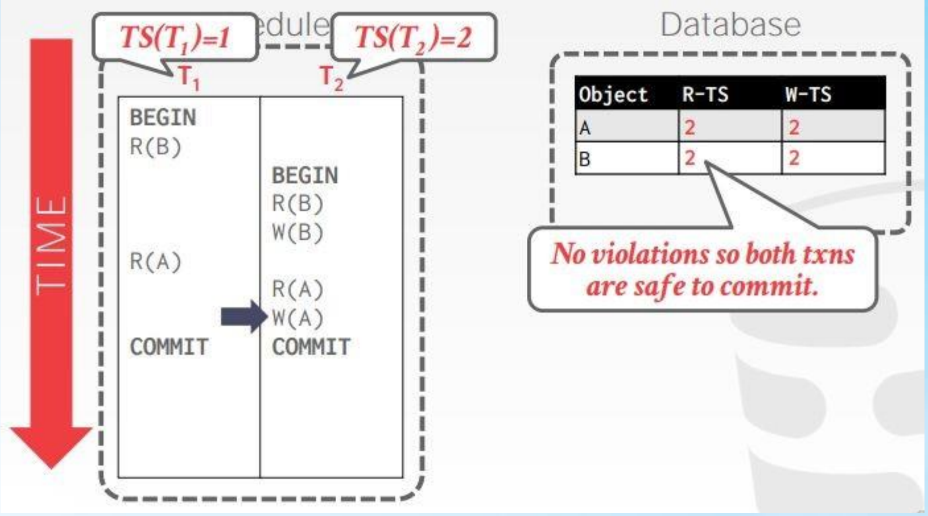
6



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

Object	R-TS	W-TS
A	2	0
B	2	2

7



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

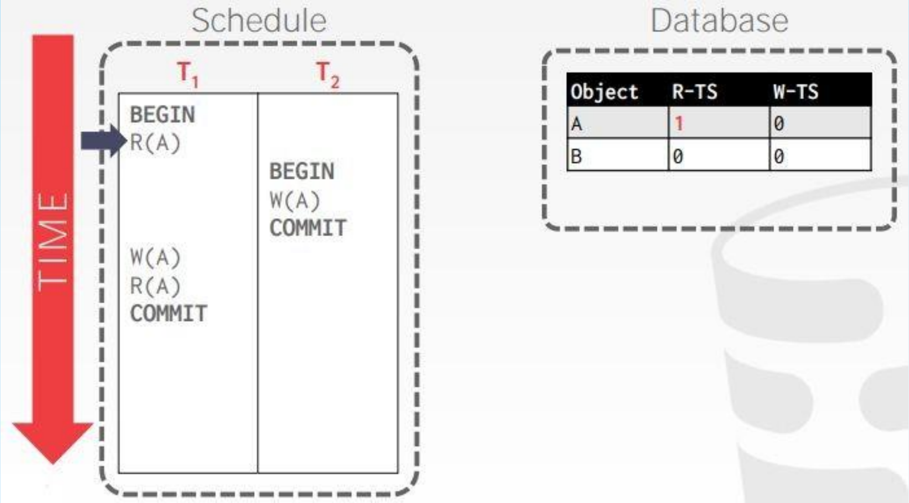
Object	R-TS	W-TS
A	2	2
B	2	2

No violations so both txns are safe to commit.

Basic Timestamp ordering protocol

Example:

1



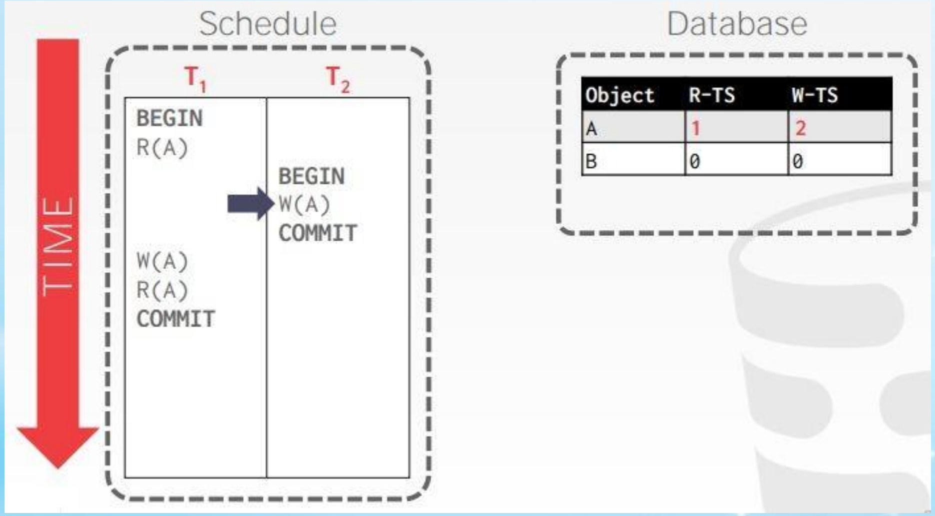
Schedule

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

Database

Object	R-TS	W-TS
A	1	0
B	0	0

2



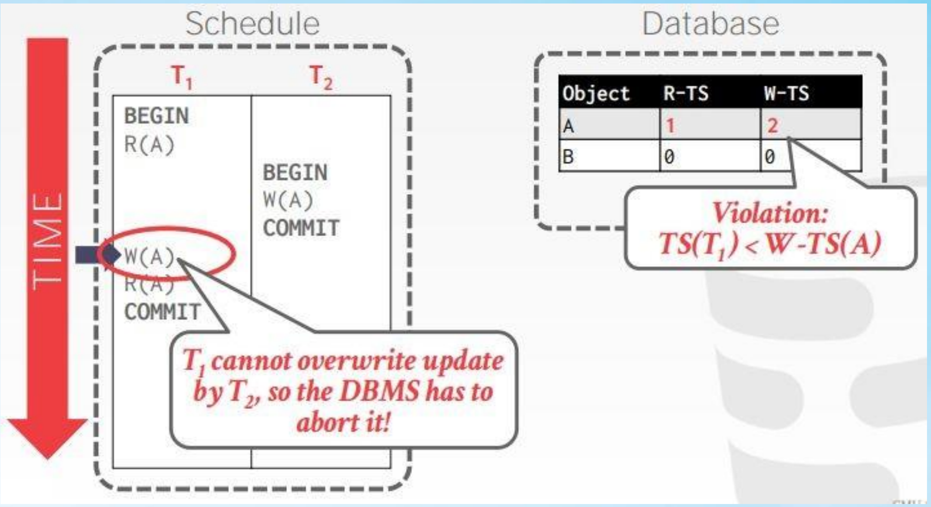
Schedule

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

Database

Object	R-TS	W-TS
A	1	2
B	0	0

3



Schedule

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

Database

Object	R-TS	W-TS
A	1	2
B	0	0

Violation: $TS(T_1) < W\text{-}TS(A)$

T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

Concurrency Control

Thomas write Rule

Thomas Write Rule provides the guarantee of Serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

The basic Thomas write rules are as follows:

- If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
- If TS(T) < W_TS(X) then don't execute the W_item(X) operation of the transaction and continue processing.
- If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).

If we use the Thomas write rule then some serializable schedule can be permitted that does not conflict serializable as illustrate by the schedule in a given figure:

In Short

If **TS(Ti)** < **R-TS(X)**:

→ Abort and restart **Ti** .

If **TS(Ti)** < **W-TS(X)**:

→ Thomas Write Rule: Ignore the write and allow the transaction to continue.

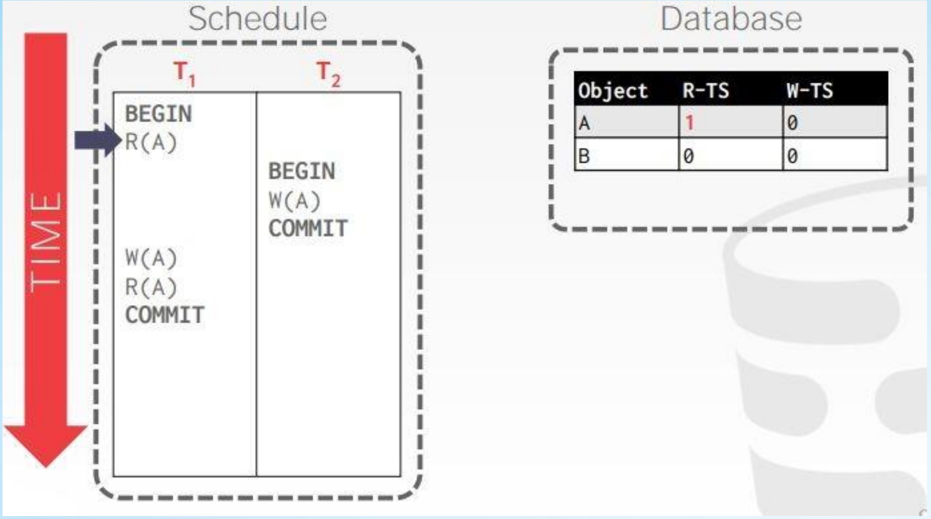
→ This violates timestamp order of **Ti** .

Else:

→ Allow **Ti** to write **X** and update **W-TS(X)**.

Concurrency Control

1



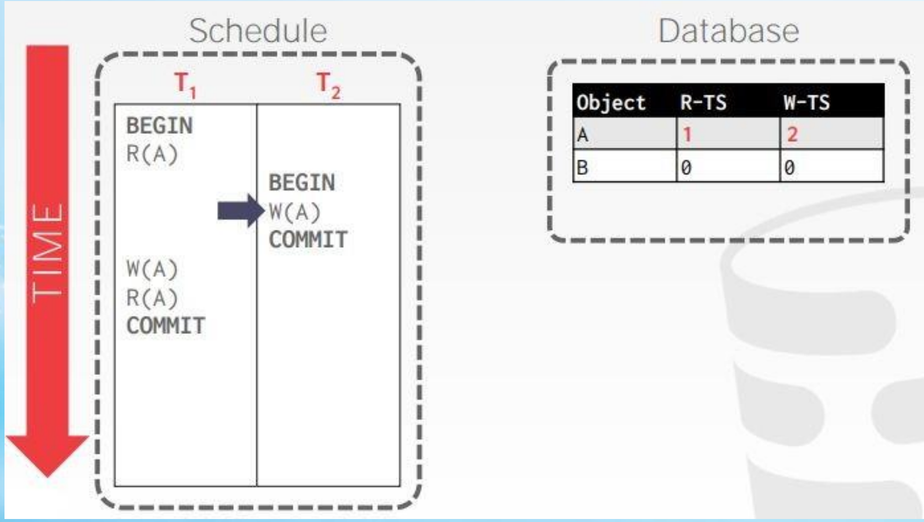
Schedule

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

Database

Object	R-TS	W-TS
A	1	0
B	0	0

2



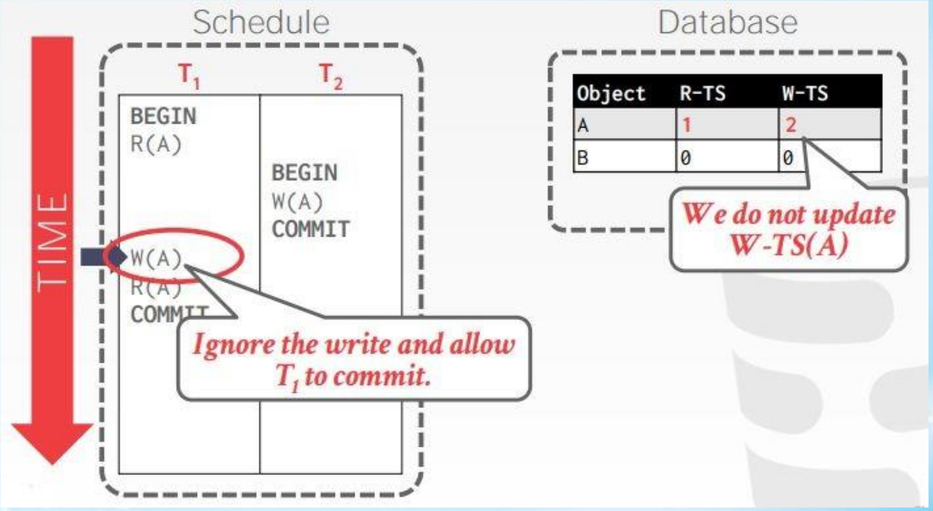
Schedule

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

Database

Object	R-TS	W-TS
A	1	2
B	0	0

3



Schedule

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

Database

Object	R-TS	W-TS
A	1	2
B	0	0

We do not update W-TS(A)

Ignore the write and allow T_1 to commit.

Concurrency Control

Validation Protocol

- A **concurrency control method** used to ensure **serializability** of transactions.
- Assumes that **conflicts between transactions are rare** — so it allows transactions to proceed without locking resources.
- Conflicts are only checked **at the end**, during a special **validation phase**, before the transaction commits.

Phases of Validation Protocol

There are **three phases**:

1. Read Phase

- The transaction reads data and makes changes to **local (temporary) variables**.
- No changes are made to the actual database yet.

2. Validation Phase

- Before committing, the system checks whether this transaction **conflicts with any other transactions** that committed during its lifetime.
- If **no conflict**, the transaction can proceed.
- If there is a **conflict**, the transaction is **aborted and restarted**.

3. Write Phase

- If validation is successful, all changes are written from temporary variables to the actual database.
- The transaction is **committed**.

Concurrency Control

Let’s look at the timestamps of each phase of a transaction:

- **Start(Tn):** It represents the timestamp when the transaction Tn starts the execution.
- **Validation(Tn):** It represents the timestamp when the transaction Tn finishes the read phase and starts the validation phase.
- **Finish(Tn):** It represents the timestamp when the transaction Tn finishes all the write operations.

This protocol uses the Validation(Tn) as the timestamp of the transaction Tn because this is actual phase of the transaction where all the checks happen. So it is safe to say that TS(Tn) = Validation(Tn).

If there are two transactions T1 & T2 managed by validation based protocol and if Finish(T1) < Start(T2) then the validation will be successful as the serializability is maintained because T1 finished the execution well before the transaction T2 started the read phase.

Concurrency Control

Example:

We have two transactions: **T1** and **T2**

- **T1** starts at time = 1 and ends at time = 4
- **T2** starts at time = 2 and ends at time = 5

They both operate on a **data item X.**

Transaction Details:

T1:

- Reads X = 100
- Prepares to write X = 150 (but doesn't write yet)

T2:

- Reads X = 100
- Prepares to write X = 200 (but doesn't write yet)

So both transactions are working on **temporary copies** of X during their **Read Phase**.

Concurrency Control

Step-by-Step Execution Using Validation Protocol:

1. T1 Read Phase

- T1 starts first
- Reads X = 100
- Plans to write X = 150
- No issues — keeps changes in temporary space

2. T2 Read Phase

- T2 starts during T1's execution
- Reads X = 100
- Plans to write X = 200
- No issues yet — still in temporary space

3. T1 Validation Phase

- T1 finishes its read phase and begins validation
- No transactions finished before T1 → so no conflicts
- T1 passes validation

4. T1 Write Phase

- T1 writes X = 150 to the database
- Transaction T1 is **committed**

Concurrency Control

Step-by-Step Execution Using Validation Protocol:

5. T2 Validation Phase

- T2 now tries to validate
- It checks: Did any **committed transaction write to X** during T2's lifetime?
- YES! T1 wrote to X while T2 was still running
  → ✗ **Conflict Detected**

6. T2 is Aborted and Restarted

- Since T1 modified a data item that T2 also accessed, T2 **cannot be serialized after T1**
- So T2 is **aborted**, and the system will **restart it later**

Transaction	Phase	Action	Status
T1	Read	Reads X	✓
T1	Validate	No earlier commit → valid	✓
T1	Write	Writes X = 150	✓ Commit
T2	Read	Reads X	✓
T2	Validate	Conflict with committed T1	✗ Abort
T2	Write	Not reached	

Concurrency Control

Deadlock

A system is in a deadlock stat if each transaction T in a set of two or more transactions is waiting for some item or resource that is locked by some other transaction T’ in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transitions in the set to release the lock on an item.

Figure shows the two transaction T1 and T2 are deadlocked in a partial schedule. T1 is on the waiting queue for item X, which is locked by T2 and T2 is on waiting queue for Y which is locked by T1.

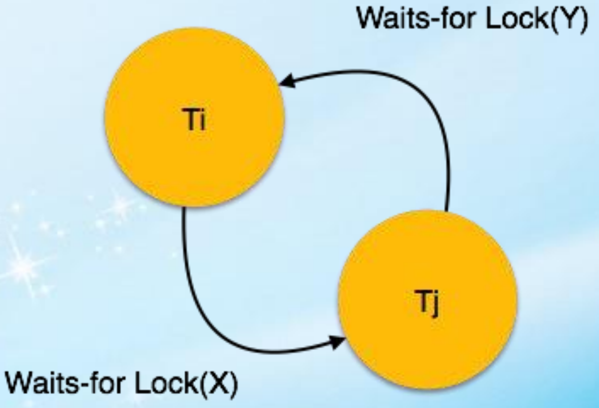
Time

T1	T2

Read_Lock(Y);
Write_Lock(Y);	

	Read_Lock(X);
	Write_Lock(X);

Write_Lock(X);	
	Write_Lock(Y);



Concurrency Control

Deadlock Avoidance

When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.

Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not.

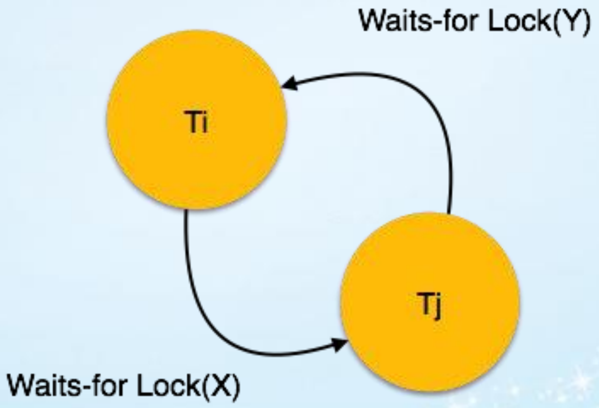
1. Wait for Graph
2. Timeout

Concurrency Control

1. Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



2. Timeout

- This method is practical because of its low overhead and simplicity.
- In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it regardless of whether a deadlock actually exists or not.

Concurrency Control

Deadlock Prevention

- To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute.
- The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur:

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock (T_i is older than T_j)− then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(T_j)$ − that is T_i is younger than T_j − then T_i is rolled back (dies). T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Concurrency Control

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

Recovery Management

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following:

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

1. Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
2. Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Recovery Management

Recovery from failure means that the database is restored to the most consistent state just before the tome of failure. To do this, the system must keep the information about the changes that were applied to data items by the various transactions. This information is kept in the System Log.

Recovery may me summarized by the following two strategies:

- **Recovery from Catastrophic Failure**

  If there is an extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method store a past copy of the database that was backed up to archival storage and reconstructs a more consistent state by reapplying or redoing the operations of committed transactions from the backed up log, up to the time of failure.

- **Recovery from non-catastrophic failures**

  When database is not physically damage but has become inconsistent due to non-catastrophic failures. The strategy is to reverse any changes caused the inconsistency by undoing some operations.

Recovery Management

There are two major techniques for recovery from non-catastrophic transaction failures:

1. Deferred updates
2. Immediate updates.

Deferred Update

- This technique does not physically update the database on disk until a transaction has reached its commit point.
- Before reaching commit, all transaction updates are recorded in the local transaction workspace.
- During commit, the updates are first recorded persistently in the log and then written to the database.
- If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed.
- It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database.
- Deferred update is also known as the **No-undo/redo algorithm.**
- It ensures transaction atomicity by recording all database modification in the log.
- All the write statements of the transactions are applied on the database only when the transaction is partially committed.
- A transaction is said to be partially committed once the final action of the transaction has been completed.

Recovery Management

Example:

The execution of transaction Ti proceeds as follows:

- Before Ti starts its execution, a record <Ti, Start> is written in the log.
- A write(X) operation by Ti results in the writing of a new record to the log as <Ti, Item, Value>.
- Finally, when Ti partially commits, a record <Ti, Commit> is written to the log.

For example, a banking system have accounts of A, B and C with initial balances 1000, 2000 and 700 respectively. We transfer 50 from account A to account B through transaction T_0. For this, we write

```
T0 :  read(A);
        A := A-50;
        write(A);
        read(B);
        B :=B+50;
        write(B);
```

Let transaction T1 withdraws 100 from account C. Then transaction T1 can be defined as

```
T1:   read(C);
        C := C-100;
        write(C);
```

Also, assume that these transactions executes serially in the order T_0 followed by T_1.

Recovery Management

Log record for these transactions will have values.

< T0 Start>
< T0, A, 950>
< T0, B, 2050)
< T0, Commit>
< T1 start>
< T1, C, 600>
< T1 Commit>

Using the log, the system can handle any failure that results in the loss of information on volatile storage.

Redo(Ti): sets the value of all data items updated by transaction Ti to the new values. The set of data items updated by Ti and their respective new values can be found in the log. The redo operation must be idempotent.

After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction Ti is redone if and only if the log contains both <Ti Start> and <Ti Commit> statements.

Recovery Management

We again consider our banking example with transactions To and T1 executed one after the other in the order To followed T1.

A	B	C
< T0 Start>	< T0 Start>	< T0 Start>
< T0, A, 950>	< T0, A, 950>	< T0, A, 950>
< T0, B, 1050)	< T0, B, 1050)	< T0, B, 1050)
	< T0, Commit>	< T0, Commit>
	< T1 start>	< T1 start>
	< T1, C, 600>	< T1, C, 600>
		< T1 Commit>

Case A:

If system fails just after the log record for the step write(B) of transaction T0 as shown in figure (A). Then, during recovery no redo operation will be done as we have only <T0 Start> in the log but not <T0 Commit>.

Case B:

If system crash occurs just after the log record write(C) as shown in figure (B). Then during recovery only T0 is done, as we have only <T0 Start> and <T0 Commit> in log disk. At the same time, we have <T1 Start> in the log but not <T1 Commit> so redo t1 will not be done.

Case C:

Similarly, if crash occurs just after the log record <T1 Commit> as shown in figure (C), the during recovery we will perform both redo(To) and redo(T1) as we have both <T0 Start> <T0 Commit> and <T1 Start> <T1 Commit> in the log disk.

Recovery Management

Immediate Update

- The database may be updated by some operations of a transaction before the transaction reaches its commit point.
- The operations are recorded forcibly in a log on disk before they are applied to the database, making recovery still possible.
- If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back by undoing the effects of its operations on the database.
- It also requires to redo the effects of committed transaction.
- Immediate Update require both undo and redo. This technique is known as **undo/redo algorithm.**
- In this method for recovery, we use the following two operations:
  - ***Undo(Ti) –*** Restores the value of all data items updated by the transaction Ti to the old values.
  - ***Redo(Ti) –*** Sets the values of all data items updated by transaction Ti to the new values.
- We need to undo a transaction T only when log contains the record <T Start> but does not contains the <T Commit>.
- *We need to do redo transaction T only when log contains the record* <T Start> and <T Commit> both.

Recovery Management

We again consider our banking example with transactions To and T1 executed one after the other in the order To followed T1.

A	B	C
< T0 Start>	< T0 Start>	< T0 Start>
< T0, A, 950>	< T0, A, 950>	< T0, A, 950>
< T0, B, 1050)	< T0, B, 1050)	< T0, B, 1050)
	< T0, Commit>	< T0, Commit>
	< T1 start>	< T1 start>
	< T1, C, 600>	< T1, C, 600>
		< T1 Commit>

Case A:

If system fails just after the log record for the step write(B) of transaction T0 as shown in figure (A). Then, during recovery we do undo(T0) operation as we have only <T0 Start> in the log but not <T0 Commit>.

Case B:

If system crash occurs just after the log record write(C) as shown in figure (B). Then during recovery, we do redo(T0) and undo(T1) as we have only <T0 Start> and <T0 Commit> in log disk. At the same time, we have <T1 Start> in the log but not <T1 Commit>. Undo(T1) should be done first than redo(T0) should be done.

Case C:

Similarly, if crash occurs just after the log record <T1 Commit> as shown in figure (C), the during recovery we will perform both redo(To) and redo(T1) as we have both <T0 Start> <T0 Commit> and <T1 Start> <T1 Commit> in the log disk.

Recovery Management

Recovery Process using Checkpoints

Recovery using the log records contains two major problems –

- The search process is time consuming & as one might conclude that recovery requires just the scanning of log as a whole for recent transactions.
- Most of the transactions that according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overheads, we introduce checkpoints.

During execution, the DBMS maintains the log, but periodically performs checkpoints consisting of the following actions –

(i) Temporarily halting the initiation of new transaction until all the active ones arc committed or aborted.

(ii) Making a backup copy of the database.

(iii) Writing all records currently residing in primary memory to stable storage.

(iv) Appending to the end to the log a record indicating that the checkpoint has occurred, then writing it to the disk.

In this a failure has occurred, a recovery process examines the log to determine the most recent transaction Ti that started executing before the most recent checkpoint took place.

For this, we search log backward from the end of log until it finds the first checkpoint record; it continue backwards until it finds the next <Ti start> record.

Recovery Management

This record identifies a transaction Ti. For example, consider the set transactions {To, T1, ……., T100} executed in the order of the subscripts. Suppose the most recent checkpoint took place during the execution of transaction T80. Thus, only transaction T80, T81, T82, …… T100 to be considered during the recovery scheme.

Each of them need to be redone if it has committed otherwise, it needs be undone.