Distributed & Mobile Computing

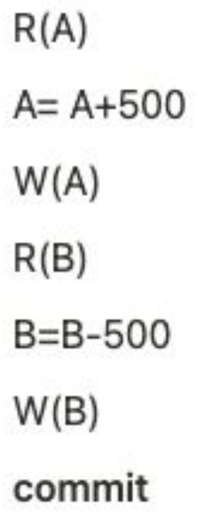
IS384

Transactions, concurrency and replication controls:

Transaction:

Transaction is a collection of related **Read** and **Write** operations used to perform some related task.

Example:



```
R(A)
A= A+500
W(A)
R(B)
B=B-500
W(B)
commit
```

Transaction Cont..

Where R(A) is a read operation to read a data item A, W(A) is a write operation to write a data item A and commit is a commit operation to commit the transaction.

ACID Properties of Transactions:

- **Atomicity**: All operations in a transaction either succeed or fail together.
- **Consistency**: The transaction transforms the database from one consistent state to another.
- **Isolation**: Concurrent transactions are isolated from each other's effects.
- **Durability**: The effects of a committed transaction are permanent.

Atomicity:

Definition: Atomicity ensures that a transaction is indivisible and is treated as a single unit of work.

Principle: Either all operations within a transaction are executed successfully, or none of them are executed at all.

Example: Consider a bank transfer involving debit from one account and credit to another. Atomicity ensures that if the debit succeeds and the credit fails (or vice versa), the entire transaction is rolled back to maintain consistency.

Consistency:

Definition: Consistency ensures that a database moves from one consistent state to another after a successful transaction.

Principle: Transactions bring the database from one valid state to another valid state, maintaining all integrity constraints, triggers, and business rules.

Example: If an operation violates a constraint (e.g., foreign key constraint), the entire transaction is rolled back, preventing any inconsistent state in the database.

Isolation:

Definition: Isolation ensures that concurrent execution of transactions does not lead to interference or inconsistency.

Principle: Transactions execute independently of each other, and their intermediate states are hidden from other transactions until they are committed.

Example: If multiple transactions are modifying the same data simultaneously, isolation ensures that each transaction sees a consistent snapshot of the data without interference from other transactions.

Durability

Definition: Durability ensures that the changes made by committed transactions persist even in the event of system failure or crash.

Principle: Once a transaction is committed, its changes are permanently stored in the database and remain intact even in the face of system failures.

Example: After a successful transaction, even if the system crashes, the changes made by that transaction are durable and will be reflected in the database when the system is recovered.

Transaction Management:

Database Management Systems (DBMS) are responsible for managing transactions and ensuring their ACID properties. This involves:

- **Concurrency control:** Preventing conflicts between concurrent transactions that access the same data.
- **Recovery:** Restoring the database to a consistent state in case of failures.
- **Locking:** Mechanisms for controlling access to shared data.
- **Logging:** Recording the actions of transactions to enable rollback and recovery.

Schedule

Schedule : A time order sequence of two or more transactions is called a schedule

For **n** transactions (T1, T2 …. Tn) , there are 2 power n total no. of possible schedules.

A schedule can also be defined as a particular sequence in which various operations of various transactions are performed.

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

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

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

Example 1: Schedule S1

T 1	T 2	T 3	T 4
R(A)			
W(A)			
	R(A)		
		W(A)	
			R(B)

Example 2: Schedule S2

T 1	T 2	T 3	T 4
R(A)			
	R(A)		
		R(A)	
			R(A)
W(B)			
	W(A)		
		W(B)	
			W(A)
			Commit
		Commit	
	Commit		
Commit			

Complete Schedule: A schedule with commit or abort operation

Concurrency Control Protocols:

- **Serializability:** A schedule of transactions is serializable if it is equivalent to some serial execution of the same transactions. This means that the results of the concurrent execution are the same as if the transactions were executed one at a time in some order.
- **Locking:** Transactions acquire locks on data items before accessing them. This prevents other transactions from modifying the data until the lock is released.
- **Optimistic concurrency control:** Transactions do not acquire locks until they are about to commit. If a conflict is detected during commit, one of the conflicting transactions is aborted and restarted.

Transaction States:

a. Active State:

- The initial state where operations within the transaction are executed.

b. Partially Committed State:

- Transaction has executed all operations successfully and is ready to be committed.

c. Committed State:

- All operations within the transaction have been successfully completed and permanently saved in the database.

d. Aborted State:

- Transaction has encountered an error or has been explicitly rolled back, and its changes are undone.

Serializability

Serializability is a concept that helps us to check which schedules are **serializable**. why we need this? - *Checking the correctness of a schedule*

A serializable schedule is the one that always leaves the database in a **consistent state.**

Types of Schedules in DBMS

- Serial schedules
- Non-Serial schedules (concurrency)

Serial schedules

A transaction is executed completely before starting the execution of another transaction. The serial schedule where one transaction must wait for another to complete all its operation This type of execution of transaction is also known as **non-interleaved execution.**

Challenges:

- Limit Concurrency
- Causes CPU waste
- Smaller transactions wait for long time

Serial Schedule example

Here R refers to the read operation and W refers to the write operation.

In this example, the transaction T2 does not start execution until the transaction T1 is finished.

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

A serial schedule doesn’t allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

Non-Serial Schedule

In the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete

A type of Scheduling where the operations of multiple transactions are interleaved.

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.

Non-serial schedule may leave the database in inconsistent state so we need to check these non-serial schedules for the Serializability.

The Non-Serial Schedule can be divided further into

1. Serializable and
2. Non-Serializable.

Serial Schedule and Non Serial Schedule

Advantage of Serial Schedule:

Never creates inconsistency , it follows the ACID properties very well.

Disadvantages:

- Poor Throughput (No. of transactions completed per unit time is less
- Poor Resource Utilization

Advantage of Non Serial (concurrence) Schedule:

- Better Throughput
- Better Resource Utilization

Disadvantages of Non Serial Schedule:

- Consistency issues may arise due to Non serialization

Converting the concurrent schedule into an equivalent schedule which has a serial order execution of its constituent transactions.

Converting the concurrent schedule into an equivalent schedule which is serializable

If this is possible then we call such a concurrent schedule as a Serializable schedule.

Conflict Pairs

R(A)		W(A)		W(A)	
	W(A)		R(A)		W(A)

Non Conflict Pairs

R(A)		R(A)		W(A)		R(A)	
	R(B)		R(A)		W(B)		W(B)

Important:
We can not swap the order of conflict pairs. As by doing so the final result changes hence the schedules no more remain equivalent.

Serializable schedule

A serializable schedule is the one that always leaves the database in consistent state.

This is used to maintain the consistency of the database.

It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not.

A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution.

However a non-serial schedule needs to be checked for Serializability.

If a schedule of concurrent ‘n' transactions can be converted into an equivalent serial schedule.

Then we can say that the schedule is serializable.

And this property is known as serializability.

In serializability, ordering of read/writes is important

(a) If two transactions only read a data item, they do not conflict and order is not important.

(b) If two transactions either read or write separate data items, they do not conflict and order is not important.

(c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Serializability is the concept in a transaction that helps to identify which non-serial schedule is correct and will maintain the database consistency.

Serializable Schedules:

- Serializable schedules are those schedules that maintain the isolation property among transactions, ensuring that their concurrent execution produces the same result as if they were executed serially (one after the other).
- These schedules prevent various anomalies such as **dirty reads**, **non-repeatable reads**, and **phantom reads** by preserving consistency and ensuring that the final state of the database is consistent.
- Serializable schedules follow the ACID properties (Atomicity, Consistency, Isolation, Durability) and ensure the highest level of isolation among transactions.

Non-Serializable Schedules:

- Non-serializable schedules are schedules that do not guarantee the same level of isolation and consistency as serializable schedules.
- These schedules may lead to concurrency-related anomalies, such as **dirty reads** (reading uncommitted data), **non-repeatable reads** (reading different values for the same data within a transaction), and **phantom reads** (new data appearing while a transaction is in progress).
- Such schedules might violate the ACID properties and can result in incorrect or inconsistent database states.

serializable

Serializable schedules in databases can be categorized into different types based on their characteristics and how they ensure serializability among concurrent transactions. These categories include:

- Conflict Serializable Schedules and
- View Serializable Schedules:

Conflict Serializable

Conflict serializability is based on the concept of conflicts between conflicting operations (read and write operations) of different transactions.

A schedule is conflict serializable if it's equivalent to a serial schedule after swapping non-conflicting operations while maintaining their original order within transactions.

Conflict serializability can be analyzed using techniques like the **precedence graph** or s**erialization graph** to determine if a schedule is conflict serializable.

View Serializable

View serializability is another form of ensuring serializability by considering the view (result set) of transactions before and after execution.

A schedule is view serializable if the final result (view) of executing concurrent transactions is equivalent to some serial execution of those transactions.

Checking view serializability typically involves analyzing the read and write sets of transactions to ensure that the final state is equivalent to some serial execution.

Conflict Serializability

one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

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

Example 1:

Operation W(X) of transaction T1 and

operation R(X) of transaction T2 are conflicting operations

because they satisfy all the three conditions mentioned above.

They belong to different transactions,

they are working on same data item X,

one of the operation in write operation.

Example 2:

Operations W(X) of T1 and

W(Y) of T2 are non-conflicting operations

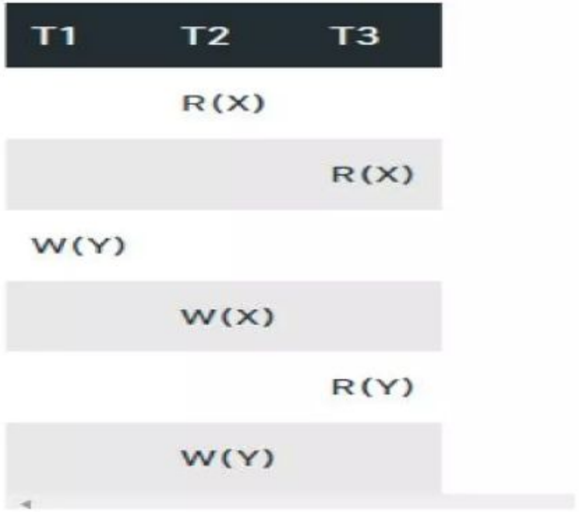
because both the write operations are not working on same data item so these operations don't satisfy the second condition.

Conflict Serializability using Precedence Graph Method

Nodes of Graph represent various Transactions

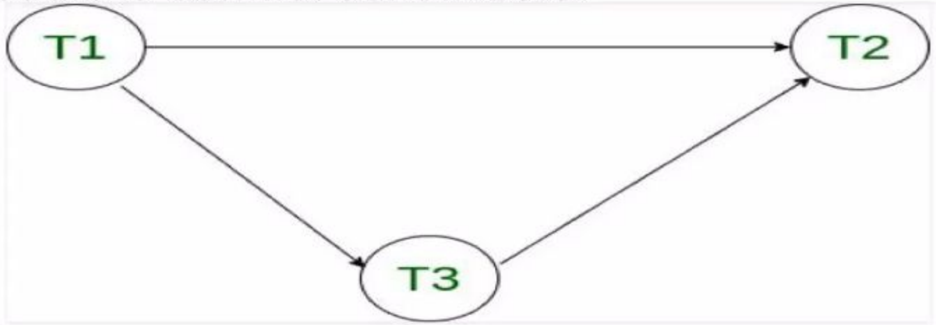
Edges represent various conflict that occurs between various transactions in a schedule.

Example of Conflict Serializability



T1	T2	T3
	R(X)	
		R(X)
W(Y)		
	W(X)	
		R(Y)
	W(Y)	

- R3(X) and W2(X) [T3 -> T2]
- W1(Y) and R3(Y) [T1 -> T3]
- W1(Y) and W2(Y) [T1 -> T2]
- R3(Y) and W2(Y) [T3 -> T2]
- Constructing the precedence graph, we see there are no cycles in the graph. Therefore, the schedule is Conflict Serializable.



$$\begin{tikzcd}
T1 \arrow[rr] \arrow[dr] & & T2 \\
& T3 \arrow[ur] &
\end{tikzcd}$$

Two operations are said to be conflicting if the belong to different transaction, operate on same data and at least one of them is a write operation.

EXAMPLE 1 : $R_1(x)\ R_1(y)\ R_2(x)\ R_2(y)\ W_2(y)\ W_1(x)$

PRECEDENCE GRAPH Examples

$$\begin{tikzcd}
T_1 \arrow[r, bend left, "R_1(y) \rightarrow W_2(y)"] & T_2 \arrow[l, bend left, "R_2(x) \rightarrow W_1(x)"]
\end{tikzcd}$$

Cyclic
↳ Hence, Non-Conflict Serializable.

Example 2 : $R_1(x)\ R_2(x)\ W_2(y)\ R_1(y)\ W_1(x)$

$$\begin{tikzcd}
T_1 & T_2 \arrow[l, "R_2(x) \rightarrow W_1(x)"'] \arrow[l, bend left]
\end{tikzcd}$$

Example 2 : $R_1(x) \underline{R_2(x)}\ \underline{W_2(y)}\ \underline{R_1(y)}\ \underline{W_1(x)}$

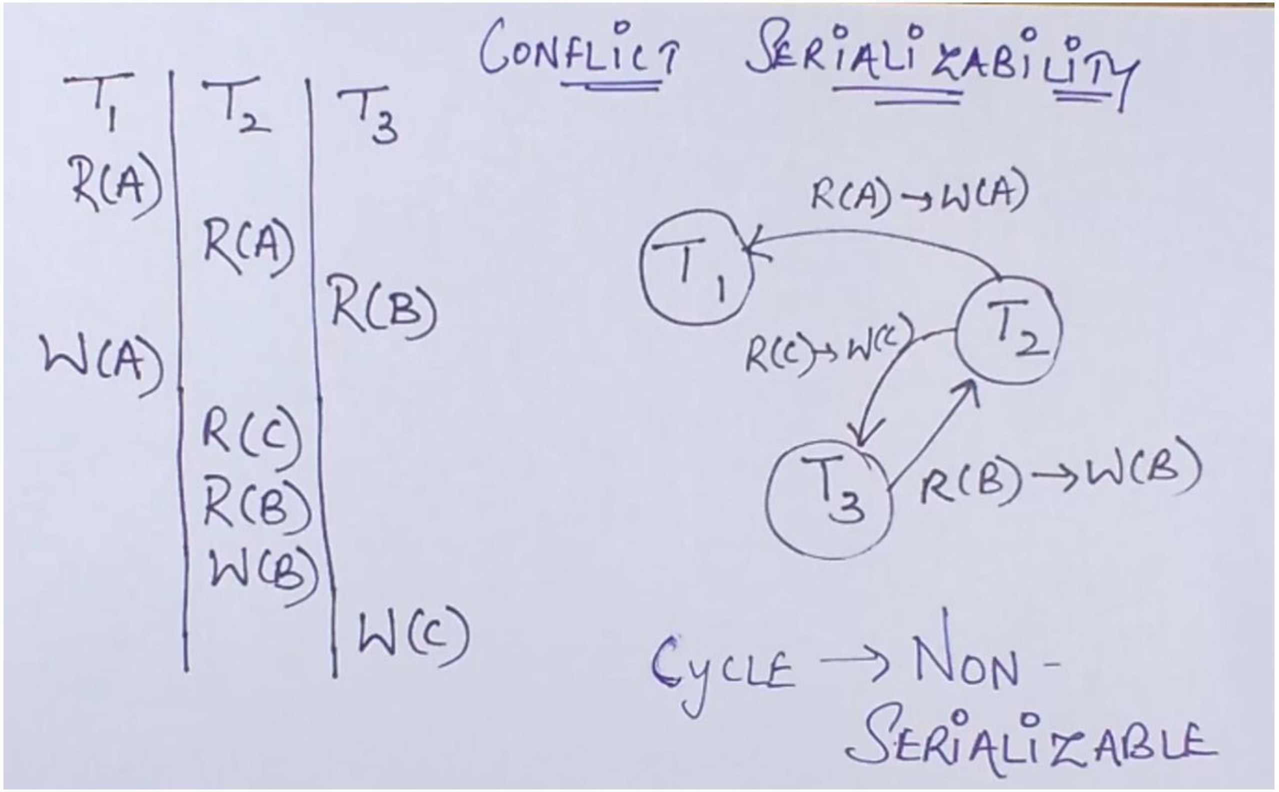
$$\begin{tikzcd}
T_1 & T_2 \arrow[l, "R_2(x) \rightarrow W_1(x)"'] \arrow[l, bend left, "W_2(y) \rightarrow R_1(y)"]
\end{tikzcd}$$

Acyclic,

Conflict Serializable.

Order : T_2 : T_1

Example

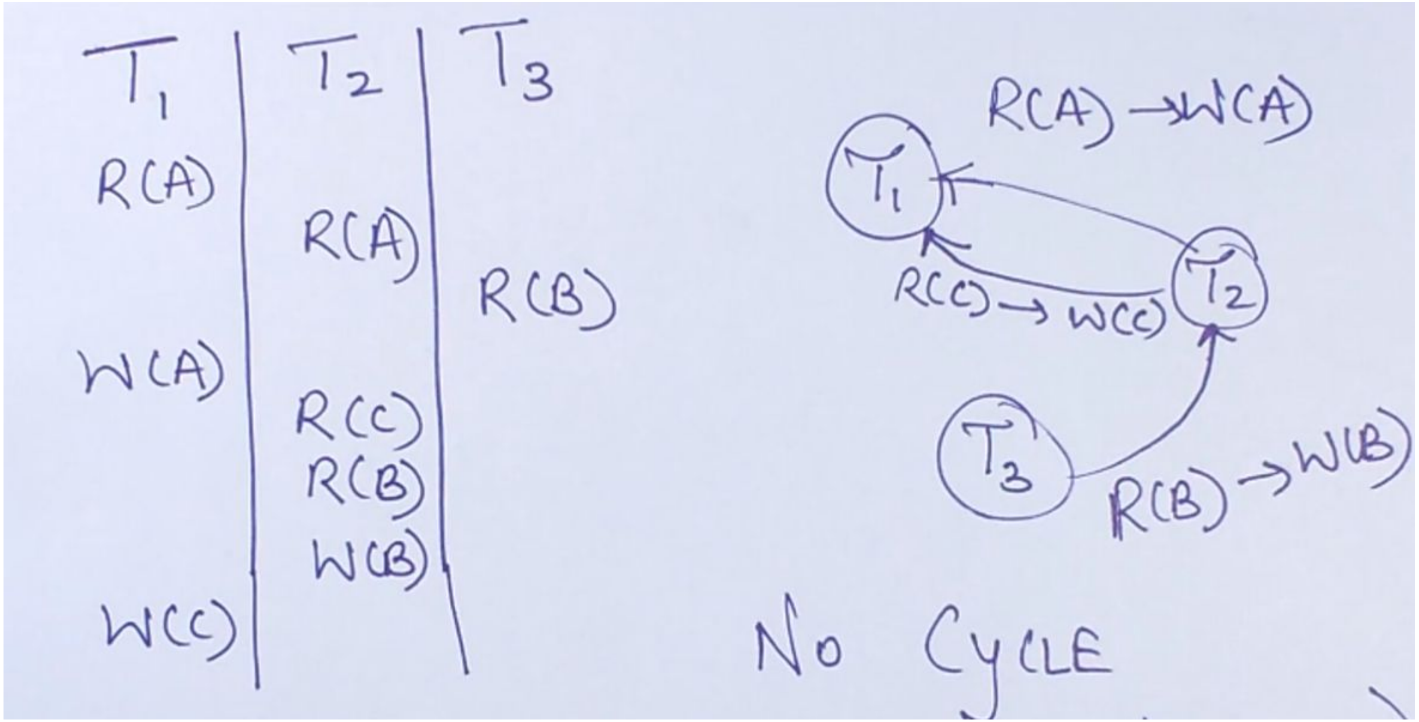


CONFLICT SERIALIZABILITY

T_1	T_2	T_3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
		W(C)

CYCLE → NON-SERIALIZABLE

Example

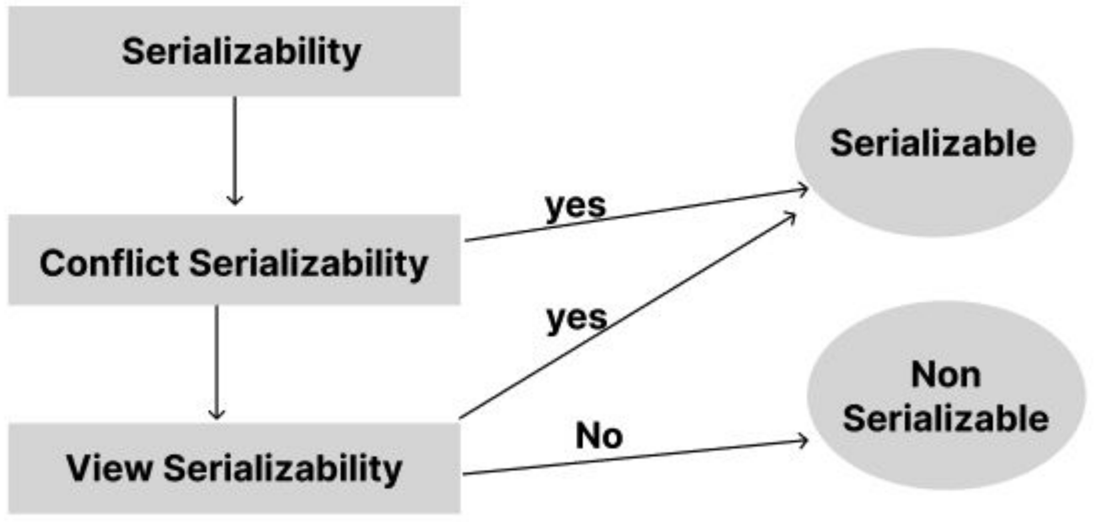


T1	T2	T3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
W(C)		

Order:

T3 -> T2 -> T1

SERIALIZABILITY: A Broader View



View Serializable

A schedule is said to be view serializable if it has an equivalent View Equivalent Schedule

View Equivalent Schedule;

A schedule S’ is said to be a view equivalent schedule to S if it satisfy the following 3 conditions

- Initial Read
- Read Write Sequence
- Final Update

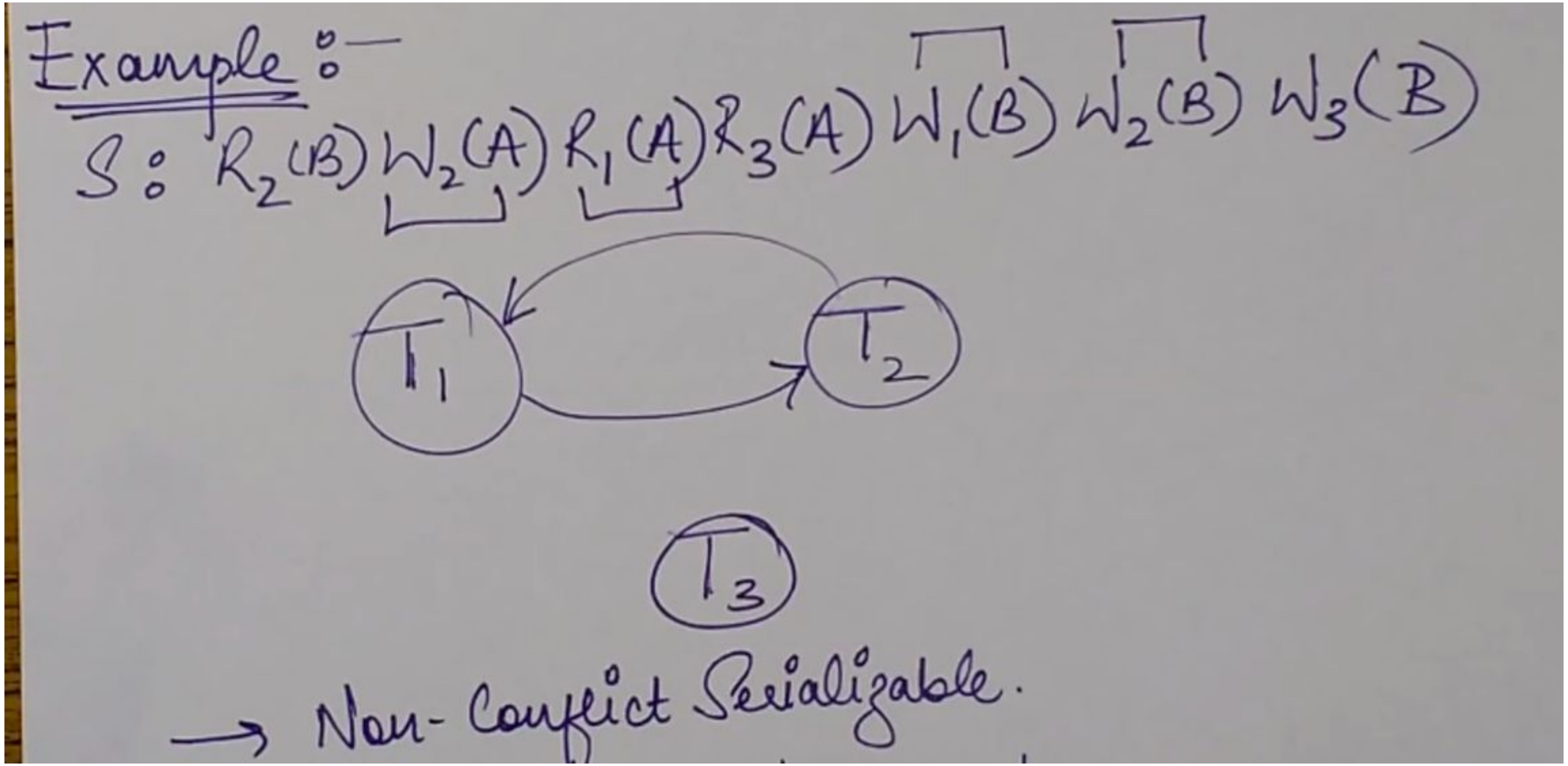
Initial Read:

Initial read -> if T1,T2….Tj transactions performs initial read (i.e Read the values of data items directly from the database before any modification (write operations) on the data items A,B,C … respectively in a schedule S, then none of these Initial Read should be violated in its view equivalent Schedule S’.

View Serializability

Read Write Sequence -> The read write conflict pairs should be in the same sequence in both the view equivalent schedules.

Final Update: If Ti is a transaction that updates a data item X finally (at the end) in a schedule S, then Ti should only update finally the data item X in the view equivalent schedule S’.

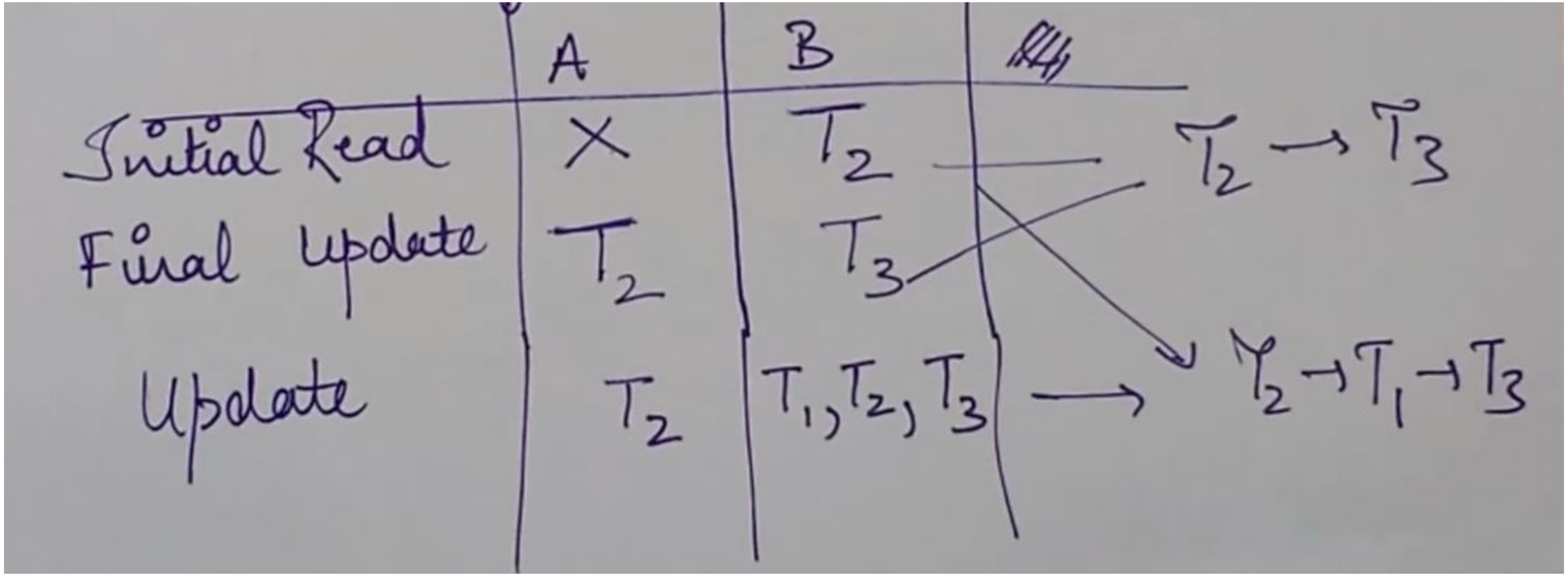


Example :-

$S: R_2(B)\ W_2(A)\ R_1(A)\ R_3(A)\ W_1(B)\ W_2(B)\ W_3(B)$

$$\begin{tikzcd}
T_1 \arrow[r, bend right] & T_2 \arrow[l, bend right] \\
& T_3
\end{tikzcd}$$

$\rightarrow$ Non-Conflict Serializable.

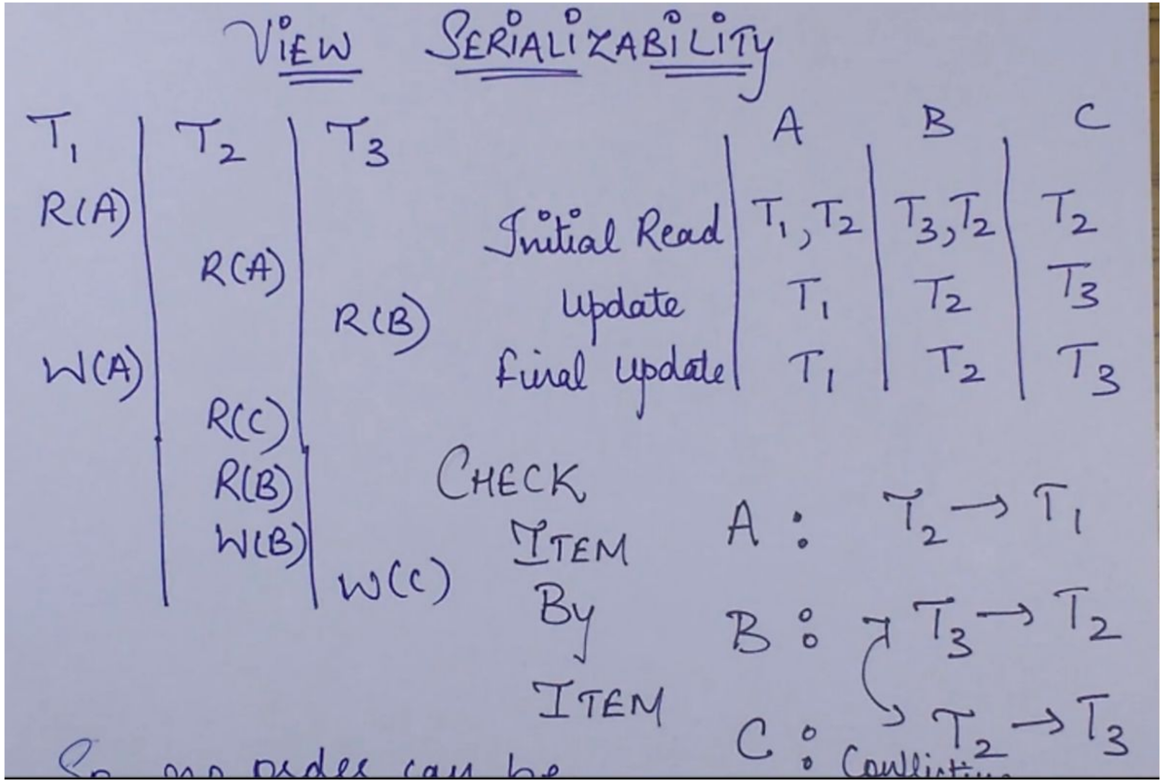


	A	B	[illegible]
Initial Read	X	T_2	
Final update	T_2	T_3	
Update	T_2	T_1, T_2, T_3	$\longrightarrow$

$T_2 \rightarrow T_3$

$T_2 \rightarrow T_1 \rightarrow T_3$

Example



VIEW SERIALIZABILITY

T_1	T_2	T_3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
		W(C)

	A	B	C
Initial Read	T_1, T_2	T_3, T_2	T_2
Update	T_1	T_2	T_3
final update	T_1	T_2	T_3

CHECK ITEM BY ITEM

A : $T_2 \rightarrow T_1$

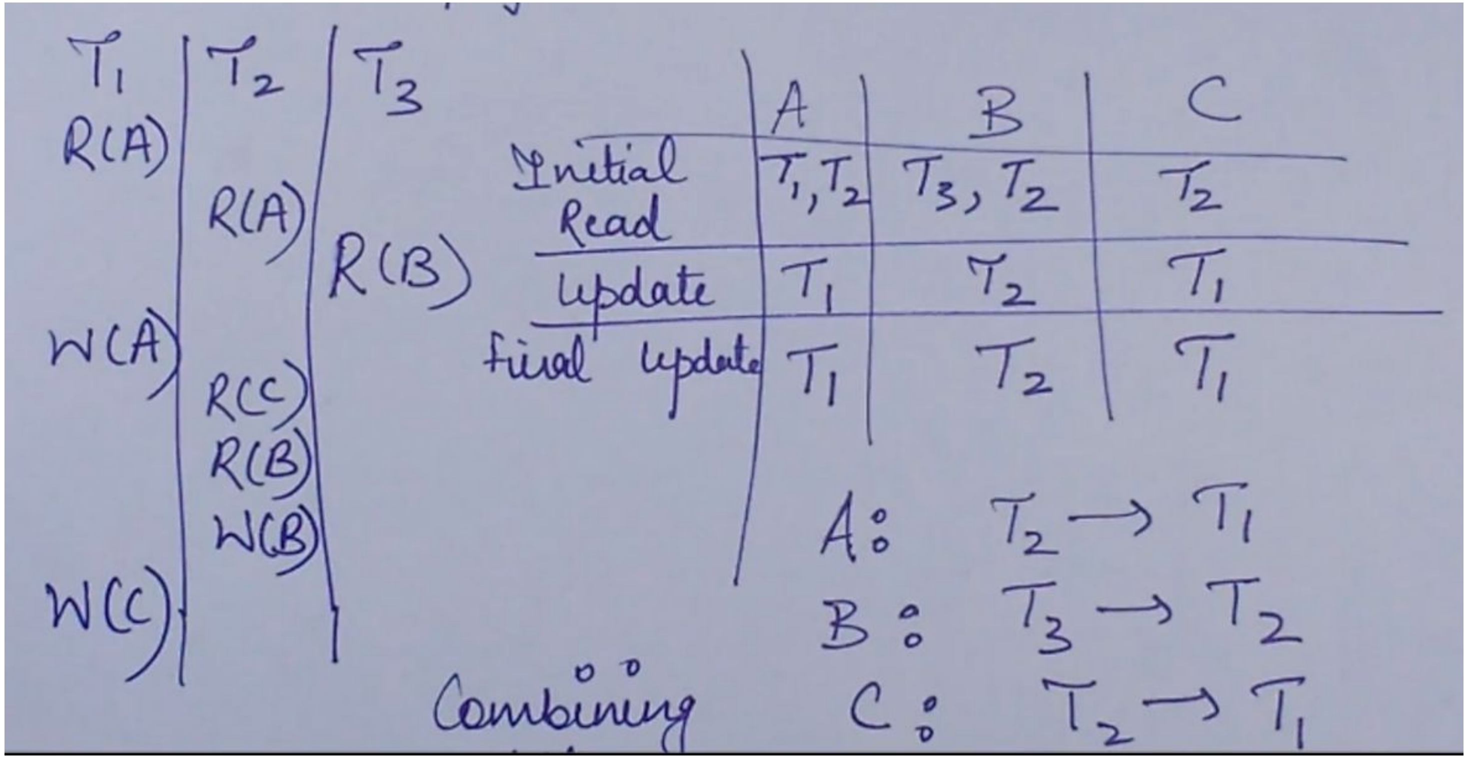
B : $T_3 \rightarrow T_2$

C : $T_2 \rightarrow T_3$

Conflicti

So no order can be

Not Serializable since B and C have conflicts.



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

	A	B	C
Initial Read	T_1, T_2	T_3, T_2	T_2
update	T_1	T_2	T_1
final update	T_1	T_2	T_1

Combining :

A : $T_2 \longrightarrow T_1$

B : $T_3 \longrightarrow T_2$

C : $T_2 \longrightarrow T_1$

Serializable now
T3 -> T2 -> T1