

Serializability and Concurrency Control in DBMS

Dr. Anish Soni

Serializability in DBMS

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

Schedules in DBMS

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

Serial Schedule

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

Serial Schedule

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

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

Non-Serial Schedule

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

Non-Serial Schedule

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

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

Types of Serializability

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



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

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

Conflicting Operations

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

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

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

Conflicting Operations

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

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

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

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

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

Conflict Serializable check

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

Example of Conflict Serializability

Lets consider this schedule:

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

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

Lets take another example:

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

Lets swap non-conflicting operations:

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

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

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

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

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

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

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

- Consider the schedule S:
- S: $r_1(x)$ $r_1(y)$ $w_2(x)$ $w_1(x)$ $r_2(y)$

Creating Precedence Graph

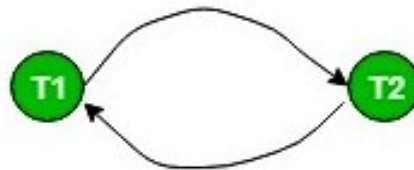
Step 1: Make two nodes corresponding to Transaction T_1 and T_2 .



Step 2: For the conflicting pair $r_1(x)$ $w_2(x)$, where $r_1(x)$ happens before $w_2(x)$, draw an edge from T_1 to T_2 .



Step 3: For the conflicting pair $w_2(x)$ $w_1(x)$, where $w_2(x)$ happens before $w_1(x)$, draw an edge from T_2 to T_1 .



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

View Serializability in DBMS

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

View Serializability in DBMS

- A schedule is view serializable if it is **view equivalent** to some serial schedule, meaning it produces the exact same final result even if the intermediate step orders differ.
- **View Equivalence Conditions:** Two schedules are view equivalent if they satisfy three rules:
 1. **Initial Read:** Each transaction must read the initial value of a data item from the same transaction in both schedules.
 2. **Updated Read (Read-Write Sequence):**
 - If transaction T_j reads a data item written by T_i in one schedule, it must do the same in the other schedule.
 - **Final Write:** The last write operation on any data item must be performed by the same transaction in both schedules.

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

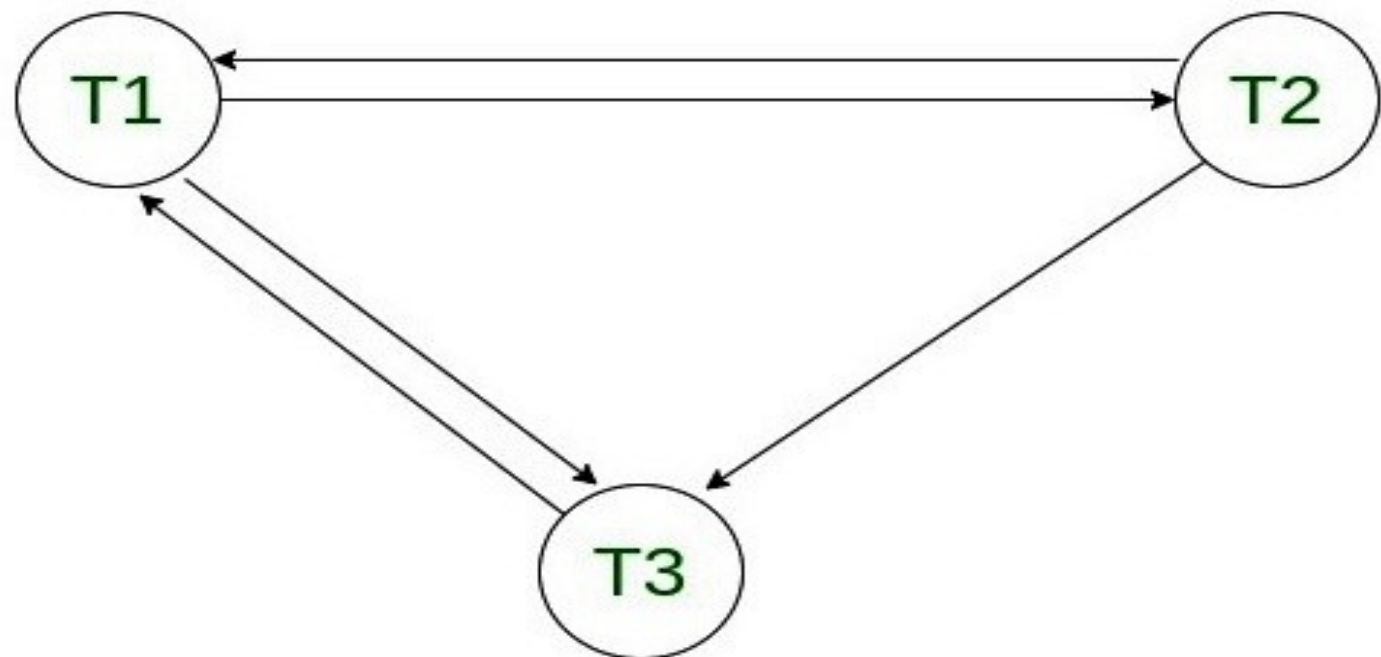
T1	T2	T3
R(A)		
	W(A)	
		R(A)
W(A)		
		W(A)

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

The conflicting operations for this schedule are -

1. $R_1(A)$ and $W_2(A)$ [$T_1 \rightarrow T_2$]
2. $R_1(A)$ and $W_3(A)$ [$T_1 \rightarrow T_3$]
3. $W_2(A)$ and $R_3(A)$ [$T_2 \rightarrow T_3$]
4. $W_2(A)$ and $W_1(A)$ [$T_2 \rightarrow T_1$]
5. $W_2(A)$ and $W_3(A)$ [$T_2 \rightarrow T_3$]
6. $R_3(A)$ and $W_1(A)$ [$T_3 \rightarrow T_1$]
7. $W_3(A)$ and $W_1(A)$ [$T_1 \rightarrow T_3$]

Constructing the precedence graph for conflicting operations in the schedule.

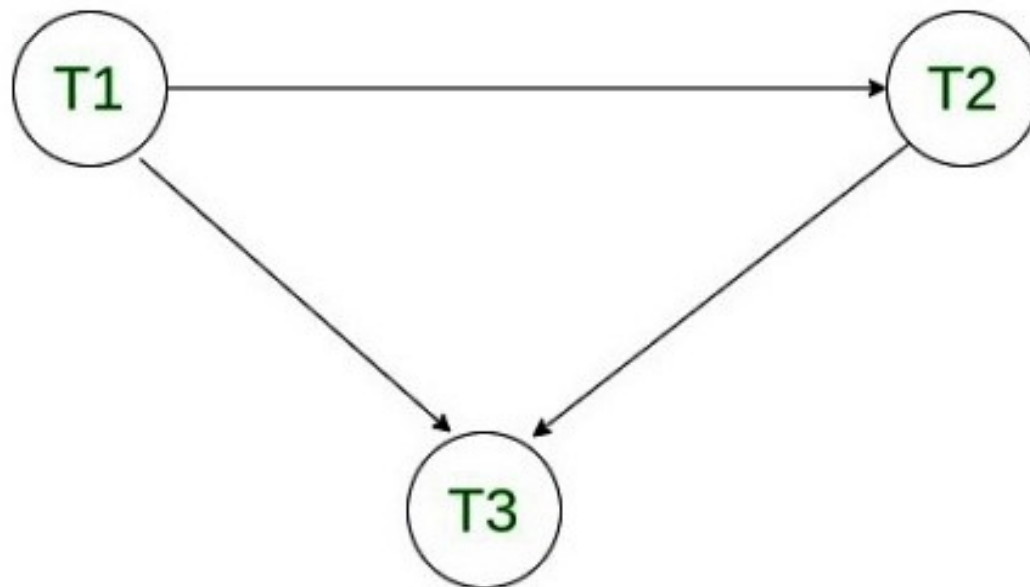


As we can see that there is a cycle in the precedence graph, it means that the given schedule is not Conflict Serializable. Now, on checking for blind write we get that there exists a blind write $W_2(A)$ in the given schedule. Thus, the schedule may or may not be View Serializable.

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

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

Constructing the dependency graph.



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

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

Introduction- Concurrency Control

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

Why Concurrency Control is Needed

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

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

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