

# Schedule

**Schedule** : It refers to the sequence in which a set of concurrent/multiple transactions are executed. You can also say it as a sequence in which the operations (such as read, write, commit, and abort) of multiple transactions are executed. It is really helpful to ensure data consistency and integrity.

If there are  $T_1, T_2, T_3, \dots, T_n$  (n) transactions then the possible schedules =  $n!$  (n factorial)

Ex : Schedule sc1

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



- **Incomplete Schedule:** An incomplete schedule is one where not all transactions have reached their final state of either commit or abort.
- **Complete Schedule:** A complete schedule is one where all the transactions in the schedule have either committed or aborted.

# Types of Schedule

## 1. Serial Schedule:

**1.Serial Schedule** : A serial schedule is one where transactions are executed one after another. We can say it like if there are two transactions T1 and T2, T1 should commit to completion before T2 starts.

Example : T1→T2

T1:Read(A)

T1:Write(A)

T1:COMMIT(T1)

T2:Read(B)

T2:Write(B)

T2:COMMIT(T2)

T2 starts only after T1 is completed/ committed.

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

### Serial Schedule

#### Advantages :

1. It follows the ACID properties. Transactions are isolated from each other.
2. It maintains consistency.

#### Challenges:

1. Since there is poor throughput (no. of transactions completed per unit time) and memory utilisation, this is not suggested as it can be inefficient.
2. Since wait time is high, less no. of transactions are completed.

## 2. Non-Serial/Concurrent Schedule:

**2. Non-Serial/Concurrent Schedule** : A non-serial schedule is one where multiple transactions can execute simultaneously (operations of multiple transactions are alternate/interleaved executions). We can say it like if there are two transactions T1 and T2, T2 doesn't need to wait for T1 to commit, it can start at any point.

Example : T1, T2, T3

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

T2 didn't wait for T1 to commit

### Non-Serial Schedule

#### Advantages :

1. Better resource utilization
2. Wait time is not involved. One transaction doesn't need to wait for the running one to finish.
3. Throughput (no. of transactions completed per unit time) is high

#### Challenges:

1. Consistency issue may arise because of non-serial execution. It requires robust concurrency control mechanisms to ensure data consistency and integrity.
2. We can use Serializability and Concurrency Control Mechanisms to ensure consistency.

### 3. Conflict-Serializable Schedule:

If the schedule can be rearranged (without violating any dependencies) to form a serial schedule, then it is conflict serializable. It's a type of Serializability.

### 4. View-Serializable Schedule:

#### 4. View-Serializable Schedule :

View serializability ensures that the database state seen by transactions in a concurrent schedule can be replicated by some serial execution of those transactions.

When we find cycle in our conflict graph, we don't know if our schedule is serializable or not, so we use the view serializability here.

A schedule is view serializable if it's view equivalent is equal to a serial schedule/execution.

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

### 5. Recoverable Schedule:

**5. Recoverable Schedule :** A recoverable schedule ensures that if a transaction reads data from another transaction, it should not commit until the transaction from which it read has committed. This helps in maintaining the integrity of the database in case a transaction fails.

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

Recoverable Schedule



## 6. Irrecoverable Schedule:

**6. Irrecoverable Schedule :** An irrecoverable schedule allows a transaction to commit even if it has read data from another uncommitted transaction. This can lead to inconsistencies and make it impossible to recover from certain failures.

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

Irrecoverable Schedule

invalid ✓

Inconsistency because if T1 make A = 10 from A = 5, then T2 comes and reads A = 10 and perform the its transaction, but when T1 fails after that. So, it rollbacks to A=5. But T2 worked on A=10 which is inconsistent data.

## 7. Cascading Schedule:

**7. Cascading Schedule :** This schedule happens when the failure or abort of one transaction causes a series of other transactions to also abort.

- T1 Writes to A: T1 writes to data item A
- T2 Reads A: T2 reads the uncommitted value of A written by T1

Now, if T1 fails and aborts, T2 must also abort because it has read an uncommitted value from T1. Consequently, T3 must abort as well

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

Cascading Schedule

Issues:

1. Performance degradation because multiple transactions need to be rolled back
2. Improper CPU resource utilisation

## 8. Cascade-less Schedule:

**8. Cascadeless Schedule :** It ensures transactions only read committed data, such that the abort of one transaction does not lead to the abort of other transactions that have already read its uncommitted changes.

- T1 Writes to A: T1 writes to data item A
- T2 Reads A: T2 reads the committed value of A

Ensuring there is no write read problem(dirty read)

Also, T2 does not read any uncommitted data, there are no cascading aborts in this schedule.

Issues :

1. Write-Write problem is encountered.
2. Performance overhead is there

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

Cascadeless Schedule

T2 waits for T1 to commit for reading A value.

It caused Write-Write problem as T1 write A to 20 from A = 10, but does not commit instead rollbacks later, but T2 reads A= 20 (made by T1) in between which leads to inconsistent data, as there is no wait for write for committed data.

Performance overhead as T2 need to wait for T1 to complete before reading A value.

## 9. Strict Schedule:

**9. Strict Schedule :** It ensures transaction is not allowed to read or write a data item that another transaction has written until the first transaction has either committed or aborted. It prevents cascading aborts.

- T2 cannot read or write the value of A until T1 has committed.
- This ensures that T2 only sees the committed value of A

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

Strict Schedule

Write-write problem is resolved. Make the most consistent schedule.



## Concurrent VS Parallel Schedule

For	Concurrent	Parallel
Defination	Concurrent scheduling manages multiple tasks at the same time. Tasks can overlap in their execution periods but do not run exactly simultaneously.	Parallel scheduling runs multiple tasks simultaneously using multiple cores or processors. Each task is executed on a separate core or processor at the same time.
Execution	Achieved on a single-core processor through context switching, where the CPU rapidly alternates between tasks, creating the illusion of simultaneous execution.	Requires a multi-core processor or multiple processors, with each core/processor handling a different task concurrently.
Example	Multi-threading on a single-core CPU, where threads take turns using the CPU.	Multi-threading on a multi-core CPU, where threads run concurrently on different cores.