Unit 4 - **Database Transaction Management**

Syllabus :-

Introduction to Database Transaction, Transaction states, ACID properties, Concept of Schedule, Serial Schedule. **Serializability**: Conflict and View, Cascaded Aborts, Recoverable and Non-recoverable Schedules. **Concurrency Control**: Lock-based, Time-stamp based Deadlock handling. **Recovery methods**: Shadow-Paging and Log-Based Recovery, Checkpoints. **Log-Based Recovery**: Deferred Database Modifications and Immediate Database Modifications.

Pt1= Introduction to Database Transactions

A **database transaction** is a **logical unit of work** that contains one or more SQL operations (such as INSERT, UPDATE, DELETE, or SELECT) performed on a database.
All the operations in a transaction are **executed as a single unit** — either **all succeed** or **none take effect**.

Why Transactions Are Important

Transactions ensure that a database remains **consistent**, **reliable**, and **accurate**, even in the event of errors, power failures, or system crashes.

For example:

- When transferring money from **Account A** to **Account B**, both the debit and credit operations must succeed.
  If one fails, the other should **not** happen.

ACID Properties of Transactions

Transactions follow four key properties — **ACID**:

Property	Description	Example
A – Atomicity	All operations in a transaction are treated as a single unit — either **all succeed** or **none**.	If money is debited from Account A but can't be credited to Account B, both operations are **rolled back**.
C – Consistency	The database must move from one **valid state** to another.	The total amount of money in the system remains the same before and after the transfer.
I – Isolation	Transactions execute **independently** of each other.	Two customers transferring money at the same time don't interfere with each other.
D – Durability	Once a transaction is **committed**, the changes are **permanent**, even if the system crashes.	After transferring money, the updated balances remain saved.

Transaction Example (SQL)

Let's take a **bank transfer** example between two accounts.

Example 1: Successful Transaction

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;  -- Debit from Account A

UPDATE accounts
SET balance = balance + 500
WHERE account_id = 2;  -- Credit to Account B

COMMIT;  -- Save the changes
```

✅ *Both updates succeed → transaction is **committed**.*

Example 2: Failed Transaction (with ROLLBACK)

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 500
WHERE account_id = 2;

-- Suppose the second update fails (e.g., account 2 doesn't exist)
ROLLBACK;  -- Undo all changes
```

Key SQL Commands for Transactions

Command	Description
BEGIN TRANSACTION	Starts a transaction
COMMIT	Saves all changes made in the transaction
ROLLBACK	Cancels the transaction and undoes changes
SAVEPOINT	Creates a checkpoint within a transaction

Example with SAVEPOINT

```
BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 1000 WHERE account_id = 1;
SAVEPOINT after_debit;
```

```
UPDATE accounts SET balance = balance + 1000 WHERE account_id = 2;


-- Something goes wrong here
ROLLBACK TO after_debit;  -- Undo from savepoint, not entire transaction


COMMIT;
```

✅ ***Summary***

- A **transaction** groups multiple database operations into a single logical unit.
- Transactions maintain **data integrity** and **consistency**.
- **ACID** properties ensure reliability in multi-user environments.
- **COMMIT** saves, **ROLLBACK** undoes.

Pt2:-Transaction states

Transaction States in DBMS

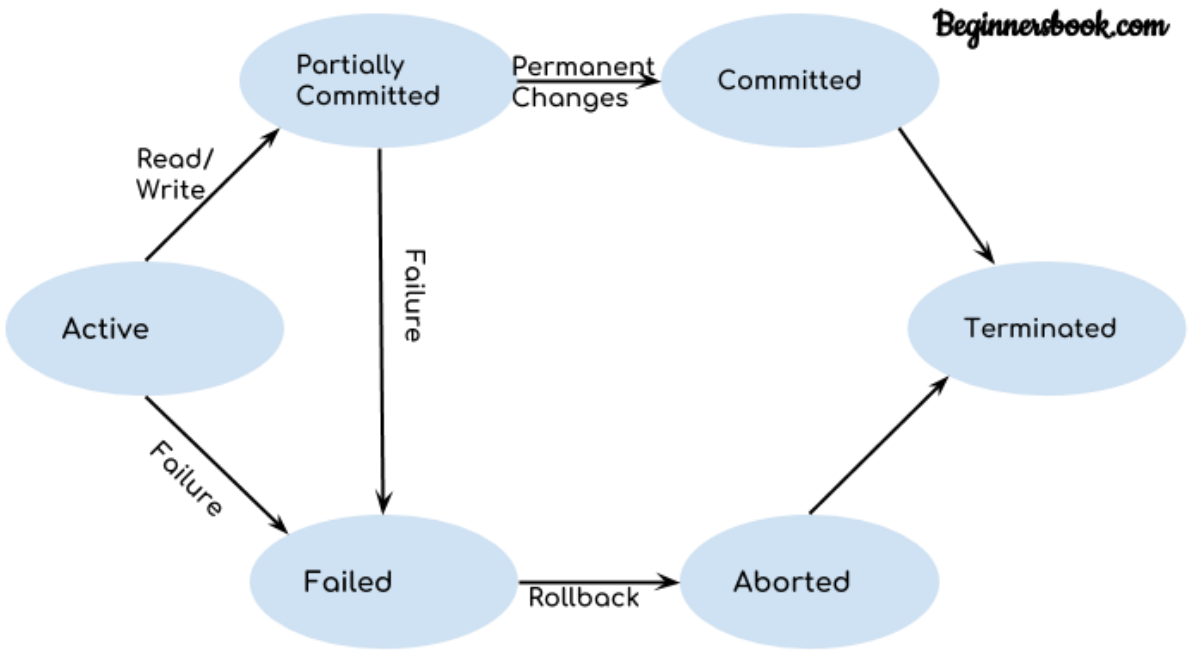
A transaction in a database goes through several states from start to finish.

Here are the main transaction states:

State	Description
1. Active	Transaction has started and operations are being executed.
2. Partially Committed	All statements have executed successfully, but changes are not yet permanently saved.

State	Description
3. Committed	Transaction is successfully completed and all changes are permanently saved.
4. Failed	Some error occurred, so the transaction cannot continue.
5. Aborted	Transaction is rolled back — changes are undone, and the database returns to its previous state.

Transaction State Diagram



Step-by-Step Explanation with Example

Let’s consider a bank transfer transaction again.

We’re transferring ₹500 from Account A to Account B.

1️⃣ Active State

Meaning:
The transaction has started, and operations are being executed.

Example:

```
BEGIN TRANSACTION;


UPDATE accounts

SET balance = balance - 500

WHERE account  id = 1;  -- Debit Account A
```

- The system is executing commands.
- The transaction is Active.

2️⃣ Partially Committed State

Meaning:
All SQL statements have executed successfully, but the final result hasn’t been committed (saved) yet.

Example:

```
UPDATE accounts

SET balance = balance + 500

WHERE account  id = 2;  -- Credit Account B
```

Now both debit and credit operations are done, but changes are still temporary.

The transaction is now Partially Committed.

3️⃣ Committed State

Meaning:
The transaction has been successfully completed — all changes are permanent in the database.

Example:

COMMIT;

Example :

- ₹500 is deducted from Account A.
- ₹500 is added to Account B.
- Data is safely written to the database.

Transaction is now in the Committed state.

4️⃣ Failed State

Meaning:
Something goes wrong — maybe a constraint violation, a system crash, or a logical error.

Example:

UPDATE accounts

SET balance = balance + 500

WHERE account_id = 2; -- Suppose Account 2 doesn't exist

An error occurs — transaction moves to the Failed state.

---

5 Aborted State

Meaning:
Since the transaction failed, all previous operations are undone (rolled back) to maintain consistency.

Example:

ROLLBACK;

The database returns to the state before the transaction started.
The transaction is now Aborted.

Summary Table

Step	Transaction State	Description	Example SQL
1	Active	Transaction is executing	BEGIN TRANSACTION
2	Partially Committed	All statements done	UPDATE ... done
3	Committed	Changes are saved	COMMIT
4	Failed	Error occurs	Constraint error, crash

Step	Transaction State	Description	Example SQL
5	Aborted	Rolled back to safe state	ROLLBACK

Key point

A transaction's journey ensures data integrity:

Active → Partially Committed → Committed
or
Active → Failed → Aborted

Pt.3. What Are ACID Properties?

In a **Database Management System (DBMS)**, transactions must maintain the **integrity** and **consistency** of data even in case of errors, failures, or crashes.

To achieve this, every transaction follows four key properties known as **ACID**:

A → Atomicity
C → Consistency
I → Isolation
D → Durability

1️⃣ Atomicity

Meaning

- Atomicity means a transaction is treated as **a single unit of work**.
- **Either all operations of the transaction are completed successfully, or none are performed at all.**
- There are **no partial changes** in the database.

Example

Suppose you are transferring ₹500 from **Account A** to **Account B**.

Transaction steps:

```
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1; -- Debit A

UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;  -- Credit B
```

If the first statement succeeds but the second fails (e.g., Account B doesn't exist),
then the entire transaction must be rolled back — Account A's balance will be restored.

✅ ***Either both debit and credit happen, or none do.***

2️⃣ Consistency

Meaning

- Consistency ensures that a transaction **transforms the database from one valid state to another**.

- The integrity **constraints** (like primary keys, foreign keys, and balance rules) must **always be satisfied**.

Example

Continuing with the bank example:

Before transaction:

- Account A = ₹2000
- Account B = ₹1000
- **Total = ₹3000**

After transferring ₹500:

- Account A = ₹1500
- Account B = ₹1500
- **Total = ₹3000**

The total amount in the system remains the same — database consistency is maintained.

If the total becomes ₹2500 or ₹3500 due to an error, consistency is broken.

3️⃣ Isolation

Meaning

- Isolation ensures that **concurrent transactions** (running at the same time) do not **interfere** with each other.

- Each transaction behaves as if it is **executed alone** on the database.

Example

Two users perform transactions at the same time:

- **Transaction 1:** Transfer ₹500 from Account A → B
- **Transaction 2:** Deposit ₹1000 into Account A

If isolation is not maintained, both transactions may read **inconsistent or partial data**, causing incorrect balances.

The DBMS uses locks or isolation levels to prevent such problems (e.g., "dirty reads" or "lost updates").

4️⃣ Durability

Meaning

- Durability ensures that once a transaction is **committed**, its changes are **permanent**, even if the system crashes immediately after.

Example

If a money transfer is committed successfully:

COMMIT;

Even if there's a **power failure** or **system crash**, the updated balances will remain saved in the database after restart.

The DBMS writes the changes to non-volatile storage (like disk or log files).

Summary Table

Property	Meaning	Ensures	Example
Atomicity	All or nothing	No partial transactions	Either both debit & credit happen or none
Consistency	Valid state → valid state	Data integrity	Total balance remains same
Isolation	Transactions don't interfere	Correct concurrent execution	No dirty reads/lost updates
Durability	Changes survive failures	Permanent results	Committed data is never lost

---

Real-Life Analogy

Imagine you're **sending a parcel** through a courier service:

Step	Analogy	Database Property
Either the parcel is sent **completely or not at all**	You don't send half a package	**Atomicity**

Step	Analogy	Database Property
The parcel should follow **rules and regulations**	No illegal content	**Consistency**
Two parcels sent simultaneously **don't get mixed up**	Separate tracking numbers	**Isolation**
Once delivered, it stays delivered	Can't "undo" delivery	**Durability**

1. What is a Schedule in DBMS?

Definition

A **schedule** is a **sequence (order)** in which the operations (like read and write) of **multiple transactions** are executed in a database system.

In other words,

A **schedule** defines **how transactions interleave** (mix together) their operations when they run **concurrently**.

Why We Need Schedules

When multiple users access a database **at the same time**, their transactions may overlap.
To maintain **consistency and correctness**, the DBMS must **control the order** of execution — this is what a **schedule** represents.

Example

Let’s take two transactions:

Transaction T1:

Read(A)

A = A - 50

Write(A)

Read(B)

B = B + 50

Write(B)

Transaction T2:

Read(A)

A = A * 1.1

Write(A)

Possible Schedules

If these two transactions are executed **concurrently**, the DBMS may mix their operations in many possible ways.

Schedule S1 (Interleaved Execution):

T1: Read(A)

T1: A = A - 50

T1: Write(A)

T2: Read(A)

T2: A = A * 1.1

T2: Write(A)

T1: Read(B)

T1: B = B + 50

T1: Write(B)

Here, operations from **T1 and T2 are interleaved** (mixed).

---

2. Types of Schedules

There are two broad types of schedules:

Type	Description
Serial Schedule	Transactions execute **one after another**, with **no interleaving**.
Non-Serial (Concurrent) Schedule	Operations from multiple transactions are **interleaved**.

---

3. Serial Schedule

Definition

A **serial schedule** is a schedule in which **only one transaction executes at a time** — no operations from another transaction occur until the first one **completes** (commits or aborts).

Formally:

A schedule is **serial** if transactions are executed **sequentially** without overlapping.

Example of Serial Schedule

Let’s consider two transactions again:

T1:

Read(A)

A = A - 100

Write(A)

T2:

Read(A)

A = A + 50

Write(A)

Serial Schedule 1 (T1 → T2):

T1: Read(A)

T1: A = A - 100

T1: Write(A)

T2: Read(A)

T2: A = A + 50

T2: Write(A)

Serial Schedule 2 (T2 → T1):

T2: Read(A)

T2: A = A + 50

T2: Write(A)

T1: Read(A)

T1: A = A - 100

T1: Write(A)

Both are **serial schedules** because the transactions execute **one after another**.

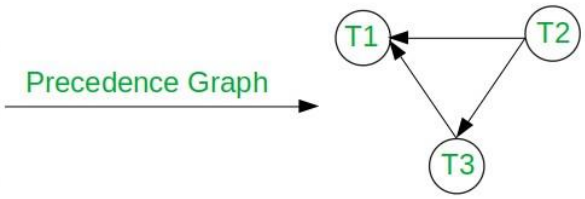
Characteristics of a Serial Schedule

- **No interleaving:** Only one transaction runs at a time.
- **Always consistent:** Since no interference occurs, data consistency is guaranteed.
- **Low concurrency:** Not suitable for multi-user systems because it reduces throughput.

Diagrammatic Representation

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

Schedule S



No overlap between transactions.

---

4. Non-Serial Schedule

A **non-serial (interleaved)** schedule allows multiple transactions to **execute simultaneously**, overlapping their operations.

This is what happens in **real-world systems** to improve performance.

Example:

T1: Read(A)

T2: Read(A)

T1: A = A - 100

T2: A = A + 50

T1: Write(A)

T2: Write(A)

*Such schedules **may lead to conflicts** (e.g., wrong final value of A) if not managed properly.*

---

5. Comparison Between Serial and Non-Serial Schedules

Feature	Serial Schedule	Non-Serial Schedule
Execution	One transaction after another	Interleaved execution
Concurrency	Low	High
Consistency	Always consistent	May be inconsistent if not controlled
Performance	Slower	Faster (if safe)
Complexity	Simple	Requires concurrency control mechanisms

---

6. Key Point:

Why Serial Schedules Matter

Even though databases rarely use serial execution (because it's slow), **serial schedules serve as the benchmark** for correctness.

If a concurrent (non-serial) schedule produces the **same result as a serial schedule**, it is said to be **serializable** — i.e., **safe and correct**.

🔸 **Example Summary**

Schedule	Type	Correctness
All of T1, then all of T2	Serial	Always correct
Mixed T1 and T2, but final result = serial result	Serializable	Correct
Mixed T1 and T2, wrong result	Non-serial & Non-serializable	Incorrect

What Is Serializability?

Definition

Serializability is the **standard for correctness** of a concurrent schedule (non-serial schedule).
A **schedule** is **serializable** if its **final effect** on the database is **the same as** that of some **serial schedule**.

In simple terms:

- Multiple transactions can run **concurrently**,
- But their combined result must be **as if they ran one after another (serially)**.

💡 **Why Serializability Matters**

Concurrency improves **performance**, but it can cause **inconsistencies** if not handled properly.
Serializability ensures:

- **Correct results**, and
- **Maximum concurrency** without violating correctness.

2️⃣ Types of Serializability

There are two main types:

Type	Definition
Conflict Serializability	Based on the **order of conflicting operations**.
View Serializability	Based on the **final read/write results** (view of data).

3️⃣ Conflict Serializability

Step 1: What Is a Conflict?

Two operations **conflict** if:

1. They belong to **different transactions**,
2. They operate on the **same data item**, and
3. **At least one** of them is a **write** operation.

Step 2: Types of Conflicts

Conflict Type	Example	Explanation
Read–Write (RW)	T1: Read(A), T2: Write(A)	Reading before writing
Write–Read (WR)	T1: Write(A), T2: Read(A)	Writing before reading

Conflict Type	Example	Explanation
Write–Write (WW)	T1: Write(A), T2: Write(A)	Both write same data

---

Step 3: Conflict Serializable Schedule

A schedule is **conflict serializable** if we can **swap non-conflicting operations** to make it **equivalent to a serial schedule**.

Example

Let’s take two transactions:

T1:

Read(A)

Write(A)

Read(B)

Write(B)

T2:

Read(A)

Write(A)

Schedule S:

T1: Read(A)

T2: Read(A)

T1: Write(A)

T2: Write(A)

T1: Read(B)

T1: Write(B)

Step 4: Build Precedence (Serialization) Graph

Rules for the graph:

- Each transaction → **a node**
- Draw an **edge** Ti → Tj if an operation in Ti **conflicts** with a later operation in Tj on the same data item.

- **Conflicts in Schedule S**

Data Item	Operation	Conflict	Direction
A	T1: Write(A), T2: Read(A)	WR conflict	T1 → T2
A	T1: Write(A), T2: Write(A)	WW conflict	T1 → T2

Graph:

T1 → T2

There is **no cycle**, so the schedule is **Conflict Serializable**.

✅ *Equivalent serial order: **T1 → T2***

Step 5: If There's a Cycle

If the precedence graph **has a cycle**,
then the schedule **is not conflict serializable**.

Example (Non-Serializable)

T1: Read(A)

T2: Write(A)

T1: Write(A)

Conflicts:

- T2 → T1 (Write–Read)
- T1 → T2 (Write–Write)

Cycle: **T1 → T2 → T1**

❌ *Not conflict serializable.*

🔷 View Serializability

Step 1: Definition

A schedule is **view serializable** if it gives the **same "view" of data** as a serial schedule —
meaning:

1. Each transaction reads the **same initial values**,
2. Each read operation reads the **same data value**, and
3. The **final writes** are on the same data items.

Step 2: When to Use It

- **View serializability** is a **broader** concept than conflict serializability.
- All **conflict-serializable schedules** are **view-serializable**,
  but not all view-serializable schedules are conflict-serializable.

Step 3: Example

Schedule S:

T1: Read(A)

T2: Read(A)

T1: Write(A)

T2: Write(A)

Conflict analysis:

- Conflicts exist → T1 → T2 → T1 (Cycle) → Not conflict serializable.

View analysis:

- T1 reads the initial value of A.
- T2 also reads the same initial value.
- The final write is by T2 (same as if T1 ran first, then T2).
  ✅ Hence, the schedule is **view serializable** (equivalent to T1 → T2).

Step 4: Difference Between Conflict and View Serializability

Feature	**Conflict Serializability**	**View Serializability**
Basis	Order of conflicting operations	Overall read/write “view”

Feature	Conflict Serializability	View Serializability
Test method	Precedence graph (no cycles)	Compare reads/writes/final writes
Complexity	Easier to test	Harder to test
Scope	Subset of View Serializability	Superset of Conflict Serializability
Cycle allowed?	No	Possibly (if same view maintained)

5️⃣ Summary Table

Concept	Meaning	Example
Schedule	Order of operations from multiple transactions	Interleaved Read, Write
Serial Schedule	Transactions run one by one	T1 → T2
Serializable Schedule	Concurrent schedule equivalent to a serial one	Same final result
Conflict Serializable	Same as serial after swapping non-conflicting ops	Tested with precedence graph
View Serializable	Same “view” of reads and writes as serial	Harder to detect but more general

---

✅ Key point:-

- **Conflict Serializability** → checks order of operations.
- **View Serializability** → checks equivalence of what transactions "see" and the final outcome.
- Both ensure the schedule behaves **as if it were serial**, preserving **data consistency**.

What Is a Cascaded Abort?

Definition

A **Cascaded Abort** (or **Cascading Rollback**) occurs when **one transaction's failure** causes **other dependent transactions** to **also fail and roll back**.

In other words —
If a transaction **uses (reads)** data written by another **uncommitted transaction**, and that first transaction **fails**, then all transactions that **depended on its data must also be undone**.

Why It Happens

This happens when the **effects of an uncommitted transaction** are made visible to other transactions — which violates **isolation**.

2️⃣ Step-by-Step Example

Let's take **three transactions**: T1, T2, and T3.

Transaction T1

Write(A)

(T1 updates data item A)

Transaction T2

Read(A)

Write(B)

(T2 reads A — which was written by T1 — and writes B)

Transaction T3

Read(B)

(T3 reads B — which was written by T2)

Schedule (Example)

Step	Operation	Description
1	T1: Write(A)	T1 modifies A
2	T2: Read(A)	T2 reads uncommitted A (from T1)
3	T2: Write(B)	T2 writes B using uncommitted data

Step	Operation	Description
4	T3: Read(B)	T3 reads uncommitted B (from T2)
5	**T1 fails**	T1 aborts

Result: Cascaded Abort

- Since **T2** read A from **T1**, T2 must **rollback** (because it used invalid data).
- Since **T3** read B from **T2**, T3 must **also rollback**.

✅ Hence, one abort (**T1**) causes a **chain reaction** of rollbacks → **Cascaded Aborts**.

If **T1** aborts, it causes:

T1 abort → T2 abort → T3 abort

Problems Caused by Cascaded Aborts

Problem	Description
Wasted Work	All dependent transactions must rollback — even if they didn't do anything wrong.
Reduced Performance	Multiple rollbacks waste CPU and I/O resources.

Problem	Description
Complex Recovery	The system must track dependencies between transactions.
Long Delays	Other users may have to wait for multiple rollbacks to complete.

4️⃣ How to Prevent Cascaded Aborts

Cascaded aborts can be **prevented** by ensuring that:

A transaction can **read only committed data**.

This concept is enforced through **isolation levels** or **scheduling protocols**.

Techniques to Prevent Cascaded Aborts

Technique	Description
Strict Schedules	A transaction **cannot read or write** data written by another transaction until it **commits**.
Cascadeless Schedules	Transactions **may read** only **committed** data items (but writes can occur).
Strict Two-Phase Locking (Strict 2PL)	Locks (especially write locks) are held until the transaction commits or aborts — ensuring no other transaction reads uncommitted data.

Recoverable and Non-recoverable Schedules.

What Is a Recoverable Schedule?

Definition

A recoverable schedule is one in which, if a transaction fails (aborts), all other transactions that have read its uncommitted data also abort — thus keeping the database in a consistent state.
In simple terms:
If one transaction depends on another, it should commit only after the transaction it depends on has committed successfully.

Why Recoverability Matters

- Ensures data consistency during failures or rollbacks.
- Prevents wrong data from being permanently saved.
- Makes database recovery easier and safer.

= Example of Recoverable Schedule

Let's take two transactions, T1 and T2.

Transactions
T1:
Read(A)
Write(A)
T2:
Read(A)
Write(B)

Schedule S1 (Recoverable)

Step	Operation	Description
1	T1: Read(A)	
2	T1: Write(A)	
3	T2: Read(A)	T2 reads the value written by T1
4	T1: Commit	T1 commits successfully
5	T2: Commit	T2 commits after T1

Explanation:

T2 read data written by T1, but T1 committed before T2.
So, even if T1 failed before step 4, T2 wouldn't have committed — ensuring consistency.

☑ **Hence, Schedule S1 is Recoverable.**

Non-Recoverable Schedule
Definition

A non-recoverable schedule is one in which a transaction commits before the transaction from which it has read data commits.
This creates a dangerous situation:
If the first transaction fails later, the dependent transaction has already committed invalid data → the database becomes inconsistent.

Example Schedule S2 (Non-Recoverable)

Step	Operation	Description
1	T1: Read(A)	
2	T1: Write(A)	
3	T2: Read(A)	T2 reads uncommitted data from T1
4	T2: Commit	Commits before T1
5	T1: Abort	T1 fails

Problem:

- T2 used uncommitted data from T1.
- Then T2 committed its changes before T1.
- When T1 aborts, the data written by T2 becomes invalid — but T2's commit cannot be undone!

❌ **Hence, Schedule S2 is Non-Recoverable.**

Diagram (Conceptual Flow)

T1: Write(A)
↓
T2: Read(A)
T2: Commit ☑
T1: Abort ❌
→ Non-Recoverable — because T2 committed before T1.

Types of Schedules (Based on Recoverability)

There are three levels of "safety" related to recoverability:

Type	Description	Cascaded Aborts Possible?	Example
Recoverable	Dependent transactions commit after the transaction they read from commits	☑ Yes	Safe but may have cascading aborts
Cascadeless	Transactions read only committed data	❌ No	No cascading aborts
Strict	Transactions can read/write only after the data's previous transaction commits	❌ No	Most restrictive and safest

Example Comparison

Type	Transaction Order (Simplified)	Behavior
Recoverable	T1: Write(A) → T2: Read(A) → T1 Commit → T2 Commit	Safe
Non-Recoverable	T1: Write(A) → T2: Read(A) → T2 Commit → T1 Abort	Unsafe
Cascadeless	T1: Write(A) → T1 Commit → T2: Read(A)	Prevents cascaded aborts

---

Key Differences Summary

Feature	Recoverable Schedule	Non-Recoverable Schedule
Dependency Rule	Dependent transactions commit only after the one they read from commits	Dependent transaction may commit before its source
Data Consistency	Maintained	May become inconsistent
Rollback Impact	Safe rollback possible	Rollback may not be possible
Cascading Aborts	May occur	Possible & dangerous
Example	T1 Commit before T2 Commit	T2 Commit before T1 Commit

---

Key point

- Recoverable Schedule → safe for rollback and recovery.
- Non-Recoverable Schedule → unsafe; should be avoided.
- Cascadeless and Strict Schedules → special types of recoverable schedules that avoid cascading rollbacks.

What Is Concurrency Control in DBMS?

Definition
Concurrency Control in DBMS is the process of **managing the simultaneous execution** of multiple transactions in a database system
so that **data integrity**, **consistency**, and **isolation** are maintained — even when many users access the database at the same time.

In Simple Words

When several users or programs try to **read or write the same data** at the same time, concurrency control ensures that:

- Transactions **do not interfere** with each other,
- The **final result** is **correct and consistent**,
- And the system performs **efficiently**.

Example
Suppose two transactions run at the same time:

T1: Transfer ₹500 from Account A to B
T2: Deposit ₹200 into Account A
Without concurrency control:

- T1 and T2 might **update A's balance incorrectly**,
- Causing **lost updates** or **wrong final result**.
-

With concurrency control:

- The database makes sure the transactions execute in a **safe order** (like one after another logically),
- So the final balance is **accurate and consistent**.

Why Concurrency Control Is Needed

Without control, concurrent transactions may cause problems such as:

Problem	Description
Lost Update	Two transactions update the same data, and one update is lost.
Temporary Inconsistency (Dirty Read)	A transaction reads data written by another uncommitted transaction.
Incorrect Summary	A transaction reads partial updates of another transaction.

Problem	Description
Unrepeatable Read	A transaction reads the same data twice and gets different results because another transaction modified it in between.

Objectives of Concurrency Control

1. **Maintain database consistency**
2. **Ensure isolation** between transactions
3. **Avoid conflicts** like lost updates and dirty reads
4. **Allow maximum parallelism** (better performance)
5. **Prevent deadlocks** and **starvation**

Methods of Concurrency Control

1. **Lock-Based Protocols**
   - Use locks to control access to data items.
   - Example: **Two-Phase Locking (2PL)** — ensures serializability.
2. **Timestamp-Based Protocols**
   - Each transaction gets a unique **timestamp** to determine order of execution.
3. **Validation (Optimistic) Protocols**
   - Transactions execute freely, then **validated** before committing.
4. **Multiversion Concurrency Control (MVCC)**
   - Keeps multiple versions of data so readers don't block writers (used in PostgreSQL, Oracle, etc.).

Summary

Aspect	Description
Definition	Managing simultaneous transactions safely
Main Goal	Consistent, correct results under concurrency
Main Issues Prevented	Lost updates, dirty reads, unrepeatable reads
Main Techniques	Locking, Timestamping, Validation
Analogy	Like traffic lights — allow smooth flow without collisions

---

Simple Analogy

Think of a **database** as a **busy road**,
and **transactions** as **cars**.
If cars move without rules, there will be **accidents (data inconsistency)**.

Concurrency control is like **traffic lights and lanes** —
it **controls the flow** so every car (transaction) reaches safely and the road (database) stays orderly.

What Is Recovery in DBMS?

Definition

Recovery in DBMS is the process of **restoring the database** to a **consistent state** after a **failure** (such as a system crash, transaction failure, or power outage).

Goal of Recovery

Ensure the **ACID properties**, especially:

- **Atomicity** → all or none of a transaction's operations are done.
- **Durability** → once committed, changes are permanent.

Types of Failures

Type	Description
Transaction Failure	Logical error (e.g., divide by zero) or user abort.
System Crash	Power loss, OS crash, etc. (memory lost, disk intact).
Disk Failure	Physical failure — data on disk is lost.

Type	Description
Media/Communication Failure	Network issues, storage corruption, etc.

2️⃣ Recovery Methods

There are two major recovery techniques in DBMS:

Method	Description
1. Log-Based Recovery	Uses a **log file** to record all transaction activities.
2. Shadow Paging	Uses **shadow copies (pages)** of data during updates.

Additionally, **Checkpoints** are used to **speed up** the recovery process.

3️⃣ Log-Based Recovery

Concept

In **Log-Based Recovery**, the system keeps a **log file** (a sequential record of all operations) on **stable storage (disk)**.

The log contains:

<Transaction_ID, Data_Item, Old_Value, New_Value>

Every change made to the database is **recorded in the log** before or after it occurs, depending on the method.

Why Logs?

If a system crash occurs:

- The DBMS can **replay the log** to **redo** committed transactions.
- Or **undo** uncommitted transactions.

Log Records Example

Type	Log Entry	Meaning
Start	<T1 Start>	Transaction T1 begins
Update	<T1, X, 100, 200>	X changed from 100 → 200
Commit	<T1 Commit>	T1 successfully committed
Abort	<T1 Abort>	T1 rolled back

4️⃣ Types of Log-Based Recovery

There are **two main approaches** based on when updates are applied to the database:

Type	Description	Redo/Undo Needed?
Deferred Database Modification	Updates are written **after commit**	Only **Redo**
Immediate Database Modification	Updates can occur **before commit**	**Redo & Undo**

A. Deferred Database Modification

Idea:
The database is **not updated immediately**.
All updates are kept in the **log** until the transaction **commits**.
Only then are the changes **applied (REDO)** to the database.

Steps:

1. Write all operations to the **log** (not to the database yet).
2. When the transaction **commits**, redo the operations from the log.

3. If a crash happens **before commit**, nothing is done (no updates on disk).

Example:

Step	Action	Database	Log
1	T1: X = X + 50	unchanged	<T1, X, 100, 150>
2	T1 Commit	Update X to 150	<T1 Commit>

✅ **If crash happens after commit → redo from log.**
❌ **If crash before commit → nothing to undo (since DB unchanged).**

Advantages:

- Simple and safe (no partial updates).
- No need for undo operations.

Disadvantages:

- Slower commit time (must write updates after commit).

B. Immediate Database Modification

Idea:

Database is **updated immediately** (before commit).
Logs are written **before each update** (Write-Ahead Logging rule).

Steps:

1. Before modifying a data item, write <T, X, old, new> to log.

2. Update the database immediately.
3. If the transaction **commits**, mark <T Commit> in the log.
4. On crash:
    - **Undo** uncommitted transactions.
    - **Redo** committed ones.

Example:

Step	Action	Log	Database
1	T1: X = X + 100	<T1, X, 200, 300>	X updated to 300
2	Crash before commit	Log available	Database may be inconsistent

After recovery:

- **Undo T1** (since no commit).

If commit had occurred before crash → **Redo T1**.

Advantages:

- Faster commits (since DB updated early).
- Better concurrency.

Disadvantages:

- More complex recovery (must handle both undo and redo).

---

5 Checkpoints

Definition

A **Checkpoint** is a mechanism used to **reduce the amount of work** the DBMS has to do during recovery.

Concept

When a checkpoint is created:

- All modified data and logs are **written to disk**.
- A special log record <Checkpoint> is added.

During recovery, the system only needs to look at transactions **after** the last checkpoint.

Checkpoint Example

<T1 Start>

<T1, X, 100, 200>

<Checkpoint>

<T2 Start>

<T2, Y, 400, 500>

System Crash

During recovery:

- Only transactions **after the last checkpoint (T2)** are checked for redo/undo.
  ✅ Saves time — no need to scan the entire log file.

Advantages of Checkpoints:

- Faster recovery time
- Reduces log scanning
- Improves performance

6️⃣ Shadow Paging

Definition

Shadow Paging is a recovery technique that maintains **two copies (versions)** of the database pages —
a **current page table** and a **shadow page table** — to ensure atomicity and durability.

How It Works

1. When a transaction starts, a **shadow page table** (a copy of the current page table) is made.
2. All updates are made on **new copies** of the pages (not the shadow pages).
3. When the transaction commits:
    - The **shadow page table** is replaced with the new page table.
4. If a crash occurs before commit:
    - The **old shadow pages** are used (database unchanged).

Example

Step	Page	Action	Result
1	Page P1	Copy to new page	New P1’ created
2	Update P1’	Changes applied	
3	Commit	Shadow table now points to P1’	
4	Crash before commit	Use old shadow P1 (safe)	

✅ **No undo/redo needed** — atomic by design.

Advantages:

- Simple to recover (just switch tables).
- No log required.

Disadvantages:

- Costly in space (duplicate pages).
- Not suitable for large databases (high I/O).
- Poor performance for frequent updates.

7️⃣ Summary Table

Method	Key Idea	Needs Log?	Redo	Undo	Notes
Deferred Modification	Apply updates after commit	✅	✅	❌	Simple & safe
Immediate Modification	Apply updates before commit	✅	✅	✅	Faster but complex
Shadow Paging	Use copies of data pages	❌	❌	❌	No log, but costly
Checkpoints	Save state periodically	✅	—	—	Speeds up recovery

8️⃣ In Simple Words

💡 **Recovery** ensures the database is correct and complete even after a crash.

- **Log-Based Recovery** → keep track of every change (undo/redo using logs).
- **Shadow Paging** → work on a copy; swap if successful.

- **Checkpoints** → quick “save points” to restart from.