**Transaction Concept (RDBMS)**

**A transaction is a sequence of one or more SQL operations executed as a single logical unit of work.**

**👉 It ensures that all operations inside it are completed successfully or none are applied.**

---

**🔸 Key Idea:**

**👉 *A transaction must follow the "all-or-nothing" principle.***

---

**🔸 Example:**

**Transfer ₹500 from Account A to Account B:**

1. **Debit ₹500 from Account A**
2. **Credit ₹500 to Account B**

**✔ Both steps must succeed together**
**❌ If any step fails → entire transaction is rolled back**

---

**🔸 Properties of Transaction (ACID)**

**1. Atomicity**

- **All operations are executed completely or not at all**

---

**2. Consistency**

- **Database remains in a valid and correct state after transaction**

---

**3. Isolation**

- **Transactions execute independently without interference**

---

**4. Durability**

- **Once committed, changes are permanent**

---

**🔸 Importance of Transaction:**

- **Maintains data integrity**
- **Ensures reliable database operations**

- **Handles system failures safely**

---

🔸 **Key Line for Exam:**

👉 ***A transaction is a logical unit of work that ensures reliable and consistent execution of database operations using ACID properties.***

---

**Transaction States (with Examples)**

A **transaction** passes through several states during its execution in a DBMS.

**1. States Overview**

- **Active** → Transaction is executing
- **Partially Committed** → Last statement executed
- **Committed** → Changes are permanently saved
- **Failed** → Error occurs
- **Aborted** → Changes are rolled back
- **Terminated** → Transaction ends

👉 **Flow:**

- Success path:
  **Active → Partially Committed → Committed → Terminated**
- Failure path:
  **Active → Failed → Aborted → Terminated**

---

**Example 1: Bank Money Transfer (Successful Transaction)**

**Transaction: Transfer ₹1000 from A to B**

**Steps:**

1. Deduct ₹1000 from Account A
2. Add ₹1000 to Account B

**State Flow:**

1. **Active**
   - Transaction starts
   - Deduct ₹1000 from A
2. **Partially Committed**
   - Credit ₹1000 to B executed
   - All statements done

3. **Committed**
    - Changes permanently saved in database
4. **Terminated**
    - Transaction ends

✅ **Final Result:** Money successfully transferred

---

**Example 2: Online Shopping Payment (Failed Transaction)**

**Transaction: Buy a product online**

**Steps:**

1. Deduct money from bank
2. Update order status

**State Flow:**

1. **Active**
    - Payment process starts
2. **Failed**
    - Network error or server crash occurs
3. **Aborted**
    - Deducted money is rolled back (refunded)
4. **Terminated**
    - Transaction ends

❌ **Final Result:** Order not placed, money returned

---

**Simple Diagram (for exams)**

Success Path:
Active → Partially Committed → Committed → Terminated

Failure Path:
Active → Failed → Aborted → Terminated

---

**Key Points to Write in Exam**

- Transaction states represent the **lifecycle of a transaction**
- Helps in **error handling and recovery**
- Ensures **data consistency and reliability**

**Easy Example: ATM Withdrawal**

**Transaction: Withdraw ₹500 from ATM**

---

**Step-by-step States**

**1. Active**

👉 You insert card and enter amount ₹500
👉 Transaction is processing

---

**2. Partially Committed**

👉 ATM checks balance and deducts ₹500
👉 Last step (cash dispensing) is about to happen

---

**3. Committed**

👉 ATM gives you cash
👉 Balance updated successfully

---

**4. Terminated**

👉 Transaction ends

---

✅ **Success Flow**

**Active → Partially Committed → Committed → Terminated**

---

**Failure Case (Easy)**

❌ **Suppose ATM runs out of cash**

**1. Active**

👉 Transaction starts

**2. Failed**

👉 ATM cannot dispense cash

**3. Aborted**

👉 Deducted money is reversed

**4. Terminated**

👉 Transaction ends

---

❌ **Failure Flow**

**Active → Failed → Aborted → Terminated**

---

**Implementation of Atomicity using Log-Based Recovery**

**1. Log-Based Recovery**

Log-based recovery is a technique used by DBMS to ensure **atomicity** by maintaining a **log file** that records all operations of a transaction.

---

🔸 **Key Points:**

- A **log file** records every change made to the database
- It is stored on **stable storage (disk)** to survive system crashes
- Log records are written **sequentially** during transaction execution

---

🔸 **Contents of Log Entry:**

Each log entry contains:

- **Transaction ID (TID)** → identifies the transaction
- **Operation performed** → read/write
- **Old value** → value before modification
- **New value** → value after modification

---

🔸 **Purpose:**

The log helps in:

- **Undo (Rollback)** → reverse changes if transaction fails
- **Redo (Recovery)** → reapply changes after commit

---

🔸 **Example:**

Transaction T1 updates balance:

- Old Balance = ₹5000
- New Balance = ₹4000

👉 Log entry:

<T1, Balance, 5000, 4000>

✔ If crash occurs **before commit** → Undo → restore ₹5000
✔ If crash occurs **after commit** → Redo → set ₹4000

---

**2. Working Flow of Atomicity Implementation**

Atomicity ensures **all-or-nothing execution** of transactions using the following steps:

🔸 **Steps:**

1. **Transaction Starts**
2. **Logging of Operations** (Write-Ahead Logging)
3. **Apply Changes to Database**
4. **Commit (if successful)**
5. **Rollback using Undo (if failure occurs)**

---

🔸 **Simple Example**

**Transaction T1:**

- Step 1: Debit ₹500 from Account A
- Step 2: Credit ₹500 to Account B

---

👉 **Case 1: Successful Execution**

✔ Both steps completed
➡ Transaction **Committed**

Final Result:

- A = A − 500
- B = B + 500

---

👉 **Case 2: Failure After Debit**

❌ Credit not executed
➡ System performs **Undo**

- ₹500 restored back to Account A

✔ Final Result:
👉 No change in database (**Atomicity maintained**)

---

🔸 **Key Line for Exam:**

👉 *Atomicity is achieved using log-based recovery, where all operations are recorded and either committed fully or rolled back completely using undo operations.*

---

### Write-Ahead Logging (WAL)

**Write-Ahead Logging (WAL)** is a technique used in DBMS to ensure **atomicity and durability**.

👉 It states that:
**"The log must be written to stable storage before the actual data is written to the database."**

---

🔸 **Key Principle:**

- **Log first → Data later**
- No database update is allowed until its **log record is safely stored on disk**

---

🔸 **Rules of WAL:**

**1. Undo Rule**

- Before modifying any data item, the **old value must be written to the log**
- Ensures that changes can be **undone** if failure occurs

---

**2. Redo Rule**

- All log records must be written to disk **before committing the transaction**
- Ensures that changes can be **redone** after a crash

---

🔸 **Working:**

1. Transaction starts
2. Changes are first written in the **log file**
3. Log is saved to **stable storage (disk)**
4. Then actual data is updated in the database
5. If successful → **Commit**
6. If failure → **Undo using log**

---

🔸 **Example:**

Transaction T1:

- Step 1: Update Account A from ₹5000 → ₹4000

👉 Log entry written first:

<T1, A, 5000, 4000>

✔ After log is saved → database is updated

---

👉 **Case 1: Crash Before Commit**

➡ Use **Undo**
➡ Restore old value = ₹5000

---

👉 **Case 2: Crash After Commit**

➡ Use **Redo**
➡ Set value = ₹4000

---

🔸 **Advantages:**

- Ensures **atomicity** (all-or-nothing execution)
- Ensures **durability** (committed data is not lost)
- Enables **recovery after system failure**

---

🔸 **Key Line for Exam:**

👉 *Write-Ahead Logging ensures that all changes are first recorded in the log before updating the database, enabling reliable undo and redo operations.*

**Implementation of Durability (RDBMS)**

**Durability** is one of the ACID properties which ensures:
👉 *Once a transaction is committed, its changes are permanently stored and will not be lost, even in case of system failure.*

---

🔸 **Techniques to Implement Durability**

**1. Stable Storage**

- Data is stored on **non-volatile memory** (e.g., hard disk/SSD)
- Unlike RAM, this storage **does not lose data after power failure**

👉 Ensures that committed data remains safe permanently

---

**2. Redo Logs**

- The DBMS maintains **log records of all changes**
- After a crash, the system uses these logs to **reapply (redo) committed transactions**

👉 Ensures committed changes are **not lost**

---

**3. Checkpoints**

- DBMS periodically saves a **consistent snapshot of the database**
- This point is called a **checkpoint**

👉 During recovery:

- System starts from the **last checkpoint** instead of the beginning
- Reduces recovery time

---

🔸 **Working Flow of Durability**

1. Transaction starts
2. Changes are written to **log file (WAL)**
3. Transaction is **committed**
4. Logs and data are saved to **stable storage**
5. In case of crash → **Redo operation** restores committed data

---

🔸 **Simple Example**

Transaction T1:

- Step 1: Update Account A ₹5000 → ₹4000

---

👉 **Case 1: After Commit**

✔ Transaction committed successfully
✔ Data stored on disk

➡ Even if **power failure occurs**,
👉 Value remains **₹4000 (not lost)**

---

👉 **Case 2: Crash After Commit but Before Data Write**

➡ Use **Redo logs**
➡ System reapplies change

✔ Final Result:
👉 Data remains **₹4000 (Durability maintained)**

---

🔸 **Key Line for Exam:**

👉 *Durability is ensured by storing data on stable storage, maintaining redo logs, and using checkpoints to recover committed transactions after failure.*

---

**Concurrent Execution (RDBMS)**

**Concurrent Execution** means:
👉 *Multiple transactions are executed simultaneously in the database system.*

---

🔸 **Advantages:**

1. **Improves Performance**
    - Multiple transactions run together → faster processing
2. **Better CPU Utilization**
    - CPU is not idle while waiting for one transaction
3. **Faster Response Time**
    - Users get quicker results

---

🔸 **Problems in Concurrent Execution:**

**1. Lost Update**

- One transaction overwrites the update of another

👉 Example:
T1 and T2 both update same account → one update is lost

---

**2. Dirty Read**

- A transaction reads **uncommitted data** of another transaction

👉 Example:
T1 updates value but not committed
T2 reads it → later T1 fails → wrong data used

---

**3. Non-Repeatable Read**

- Same data read twice gives **different values**

👉 Example:
T1 reads value

T2 updates and commits
T1 reads again → value changed

---

**4. Phantom Read**

- New rows appear/disappear during transaction

👉 Example:
T1 queries records
T2 inserts new record
T1 queries again → extra row appears

---

🔸 **Working Concept:**

- Transactions execute **simultaneously**
- DBMS uses **scheduling and isolation techniques** to manage conflicts

---

🔸 **Simple Example:**

Two users accessing same account:

- User 1 (T1): Withdraw ₹500
- User 2 (T2): Deposit ₹1000

👉 If executed concurrently without control:
➡ Conflict may occur (wrong balance)

---

🔸 **Key Line for Exam:**

👉 *Concurrent execution allows multiple transactions to run simultaneously, improving performance but may lead to problems like lost update, dirty read, non-repeatable read, and phantom read.*

---

**Seriali**

**zability (RDBMS)**

**Serializability** ensures:
👉 *Concurrent execution of transactions produces the same result as some serial (one-by-one) execution.*

---

🔸 **Types of Serializability:**

**1. Conflict Serializability**

- Based on **conflicting operations** (Read-Write, Write-Write, Write-Read)

- If a schedule can be transformed into a serial order → it is conflict serializable

---

**2. View Serializability**

- Based on **final result (view)** of transactions
- Less strict than conflict serializability

---

🔸 **Why Needed:**

- Maintains **data consistency**
- Avoids incorrect results due to concurrency

---

🔸 **Example:**

Transaction T1 and T2:

- T1: Read(A), Write(A)
- T2: Read(A), Write(A)

👉 Serial execution:

- T1 → T2 OR T2 → T1

👉 Concurrent execution must give **same result as one of these**

---

🔸 **Key Line for Exam:**

👉 *Serializability ensures that concurrent transactions produce results equivalent to serial execution.*

---

**7. Isolation (RDBMS)**

**Isolation** ensures:
👉 *Transactions execute independently without interference from other transactions.*

---

🔸 **Isolation Levels:**

**1. Read Uncommitted**

- Allows dirty reads

**2. Read Committed**

- Prevents dirty reads

**3. Repeatable Read**

- Prevents non-repeatable reads

**4. Serializable**

- Prevents all concurrency problems

---

🔸 **Purpose:**

- Prevents:
  - Dirty Read
  - Lost Update
  - Non-repeatable Read
  - Phantom Read

---

🔸 **Example:**

- T1 updates data but not committed
- T2 should **not see this change** (high isolation level)

---

🔸 **Key Line for Exam:**

👉 *Isolation ensures that transactions are executed as if they are running alone in the system.*

---

**8. Recoverability (RDBMS)**

**Recoverability** ensures:
👉 *A transaction commits only after all transactions whose changes it has read are committed.*

---

🔸 **Why Needed:**

- Prevents **inconsistent database state**
- Ensures safe recovery after failure

---

🔸 **Types of Recoverability:**

**1. Recoverable Schedule**

- T2 commits **after** T1 commits (if T2 read T1's data)

---

**2. Cascading Rollback**

- If one transaction fails → dependent transactions also rollback

---

**3. Cascadeless Schedule**

- Transactions read only **committed data**
- Avoids cascading rollback

---

**4. Strict Schedule**

- No transaction can read/write data until previous transaction commits

---

🔸 **Example:**

- T1 updates A but not committed
- T2 reads A

👉 If T1 fails → T2 must also rollback (**cascading rollback**)

---

🔸 **Key Line for Exam:**

👉 *Recoverability ensures that transactions commit in a safe order to maintain database consistency and avoid cascading failures.*