1. Transaction Concepts

A **transaction** is a logical unit of work that consists of one or more database operations (read/write) executed as a single, indivisible unit.

The SQL Transaction Process

Transactions are controlled using specific SQL commands. The most common commands used for managing transactions are:

1. **BEGIN TRANSACTION:** Starts a transaction.

Example: `BEGIN TRANSACTION TransferFunds;`

2. **COMMIT:** Commits the transaction, saving all changes made.

Example:
```
DELETE FROM Student WHERE AGE = 20;

COMMIT;
```

3. **ROLLBACK:** Rolls back the transaction, undoing all changes made since the BEGIN TRANSACTION.

Example:
```
DELETE FROM Student WHERE AGE = 20;

ROLLBACK;
```

4. **SAVEPOINT:** Marks a point within a transaction to which you can later roll back.

Example:
```
SAVEPOINT SP1;

//Savepoint created.
DELETE FROM Student WHERE AGE = 20;
//deleted
```

```
SAVEPOINT SP2;

//Savepoint created.
```

ROLLBACK TO SAVEPOINT Savepoint_Name;

RELEASE SAVEPOINT Savepoint_Name;

Transactional SQL Statement using **Explicit Transaction Control** in **SQL (Structured Query Language)**.

Example 1:

```
BEGIN TRANSACTION;
   UPDATE Accounts SET balance = balance - 500 WHERE acc_no = 'A1';  -- Debit
   UPDATE Accounts SET balance = balance + 500 WHERE acc_no = 'A2';  -- Credit
COMMIT;
```

If any operation fails, the entire transaction is rolled back to maintain consistency.

Example 2(Bank Transfer In Detail):

```
BEGIN TRANSACTION;

-- Deduct $150 from Account A
UPDATE Accounts
SET Balance = Balance - 150
WHERE AccountID = 'A';

-- Add $150 to Account B
UPDATE Accounts
SET Balance = Balance + 150
WHERE AccountID = 'B';

-- Commit the transaction if both operations succeed
COMMIT;
```

2. Properties of Transactions (ACID)

ACID properties ensure **data integrity, reliability, and correctness** in a **transactional database system**. These properties are **essential for maintaining database consistency, especially in concurrent transactions and failure scenarios**.

📌 1. Atomicity (All or Nothing)

Definition:

Atomicity ensures that a **transaction is treated as a single, indivisible unit**. Either **all operations of the transaction are executed**, or **none of them occur**.

Example:

Consider a **bank transfer of $500 from Account A to Account B**:

BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 'A';
UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 'B';

COMMIT;

💡 **Scenario:**

- If the **first UPDATE succeeds** (money is deducted from A) but the **second UPDATE fails** (due to a crash or network failure), the transaction **must be rolled back**.
- Without **atomicity**, Account A would lose money without Account B receiving it.

✅ **Atomicity ensures that either both operations happen or none happen (Rollback on failure).**

📌 2. Consistency (Valid State Before & After)

Definition:

Consistency ensures that **a transaction transforms the database from one valid state to another valid state**, maintaining **all integrity constraints**.

Example:

Consider a **bank balance rule: An account cannot have a negative balance**.

```
BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 'A';
UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 'B';

IF (SELECT Balance FROM Accounts WHERE AccountID = 'A') < 0
  ROLLBACK;
ELSE
  COMMIT;
```

💡 **Scenario:**

- If **Account A has only $300**, deducting $500 would make the balance **negative**, violating consistency.
- The transaction **rolls back** to maintain consistency.

✅ **Consistency ensures that database rules and constraints are never violated.**

📌 3. Isolation (Concurrent Transactions Do Not Interfere)

Definition:

Isolation ensures that **concurrent transactions execute independently without interfering with each other**.

Example:

Two transactions **T1 (Withdraw $500 from A)** and **T2 (Check balance of A)** occur simultaneously.

Without Isolation (Dirty Read Problem):

T1:

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 'A';

T2:

SELECT Balance FROM Accounts WHERE AccountID = 'A'; -- Reads incorrect data

- If T2 **reads Account A's balance while T1 is still in progress**, it may see an **intermediate state** (before commit).
- If T1 **fails and rolls back**, T2 would have used an **incorrect balance**.

✅ **Isolation ensures that T2 waits for T1 to complete (via locking mechanisms like Read Committed, Serializable isolation levels).**

📌 4. Durability (Permanent Changes)

Definition:

Once a **transaction is committed**, its changes **must be permanently stored**, even if the system crashes.

Example:

Consider a **ticket booking system**:

BEGIN TRANSACTION;

UPDATE Tickets SET Status = 'Booked' WHERE TicketID = 101;

COMMIT;

💡 **Scenario:**

- If a power failure occurs **after the COMMIT**, the changes **must still persist** when the system restarts.

- **Database logs (WAL - Write Ahead Logging) ensure durability by saving committed transactions to non-volatile storage.**

✅ **Durability ensures that once committed, transactions survive failures.**

📌 Summary Table of ACID Properties

Property	Definition	Example Scenario
Atomicity	A transaction is either fully completed or fully rolled back	Bank transfer: Deduct money from A **and** add to B, or rollback both
Consistency	The database remains valid before and after a transaction	A bank account cannot go negative
Isolation	Concurrent transactions execute independently	One transaction updating a balance doesn't interfere with another reading it
Durability	Committed transactions are permanent	A booked movie ticket remains confirmed even after a system crash

---

🚀 Why Are ACID Properties Important?

✔ **Prevents data corruption and loss**
✔ **Ensures correctness in multi-user environments**
✔ **Maintains integrity even in failures**
✔ **Essential for banking, e-commerce, and enterprise systems**

3. Serializability of Transactions

Serial Schedule: Transactions execute one after another (no interleaving).
Serializable Schedule: A non-serial schedule that produces the same result as some serial schedule.

Types of Serializability:
1. Conflict Serializability:
  - Two operations conflict if they:
    - Belong to different transactions.
    - Operate on the same data item.
    - At least one is a writer.

Example:

T1: R(A), W(A)
T2: R(A), W(A)

- Non-serial but conflict-serializable if equivalent to T1 → T2 or T2 → T1.

2. View Serializability:
  - Less strict than conflict serializability.
  - Allows blind writers (writes without reading first).

4. Testing for Serializability

Method: Precedence Graph
1. Construct a graph where:
  - Nodes = Transactions (T1, T2, ...).
  - Edge Ti → Tj if an operation in Ti conflicts with and precedes an operation in Tj.

2. Check for cycles:
  - If there are no cycles, the schedule is conflict-serializable.

Example:
Schedule S:

T1: R(A), W(A)
T2: R(A), W(A)
T3: R(B), W(B)

Conflicts:
  - T1 → T2 (both access A, T1 writes first)
  - No conflict between T3 and others.
- Graph: T1 → T2 (acyclic) ⇒ Conflict-serializable.

Summary
✔ Transactions ensure logical grouping of operations.
✔ ACID properties guarantee reliability.
✔ Serializability ensures correct concurrent execution.
✔ Precedence graph helps test conflict serializability.

Next Lecture: Concurrency Control Techniques (Locking, Timestamping, Optimistic Methods)