**Day – 5**

* Triggers, Cursors
* Transactions & ACID
* Indexing

**🔁 TRIGGERS — *"Auto Reaction"***

**📌 What is a Trigger?**

A **trigger** is like an **automatic alarm** or **reaction** that runs **when something happens** in your table — like **Insert**, **Update**, or **Delete**.

💡 You don’t run a trigger manually — **it runs by itself** when a rule is met.

**🧠 Real-Life Example:**

| **🔍 Action on Table** | **🔁 Trigger Will Do** |
| --- | --- |
| A new row is inserted | Send alert, log it, update another table |
| A row is updated | Automatically check or fix data |
| A row is deleted | Archive data before deletion |

-- Trigger to log every new employee added

CREATE TRIGGER trg\_EmployeeInsert

ON Employees

AFTER INSERT

AS

BEGIN

INSERT INTO EmployeeLog(Name, ActionDate)

SELECT Name, GETDATE() FROM inserted;

END

🧠 So whenever a new employee is added → this trigger will automatically log it into EmployeeLog

**✅ Types of Triggers in SQL**

There are **3 main types** of triggers:

**🔹 1. AFTER Trigger (Most Common)**

✅ **Runs *after*** an INSERT, UPDATE, or DELETE happens.

📌 Use it when you want to take action **after the data is successfully changed**.

🧠 Use Case: Logging, sending notification, auditing changes.

**🔹 2. INSTEAD OF Trigger**

🔄 Replaces the default INSERT/UPDATE/DELETE action with your **custom logic**.

📌 Use it when you want to **control or stop the change** or modify it in a special way.

🧠 Use Case: Soft delete (move data to another table), validation, preventing changes.

➡ It **won’t delete** the row.  
➡ It will just **change the status to 'Inactive'** like this:

You don’t want to allow real DELETE.  
Instead, when someone runs DELETE, you want to **just update the status to "Inactive"**.

**💥 Who Triggers the "INSTEAD OF DELETE"?**

➡ The **trigger activates automatically** when **someone runs a DELETE statement** on the table.

But instead of deleting, your **custom logic runs** — like changing the status to 'Inactive'.

🚫 **SQL will NOT delete the row.**

**🔹 3. DDL Trigger (Database-Level)**

⚙️ Fires on **schema-level events**, like CREATE, ALTER, DROP (not data).

📌 Use it to **control or track changes to tables, users, databases**.

🧠 Use Case: Prevent dropping tables, tracking schema changes, security monitoring.

**Important -** Unlike other triggers, **DDL Triggers do not deal with data changes** — they deal with **schema changes**.

**⚙️ Example Use Cases:**

| **🔧 Task** | **🧠 What DDL Trigger Can Do** |
| --- | --- |
| Prevent table from dropping | Block accidental DROP TABLE |
| Monitor schema changes | Log who changed what and when |
| Alert on database changes | Send an alert when someone alters tables |
| Enforce policies | Stop users from creating tables in production |

**🔁 Three Key SQL Trigger Types – With Purpose**

| **🔹 Trigger Type** | **🔍 What It Works On** | **⚙️ When It Fires** | **✅ Use Case Example** |
| --- | --- | --- | --- |
| ✅ **INSTEAD OF Trigger** | **Data level (DML)** – INSERT, UPDATE, DELETE | **Before** the action happens → **replaces it** | Soft delete (mark inactive instead of deleting) |
| ✅ **AFTER Trigger** | **Data level (DML)** – INSERT, UPDATE, DELETE | **After** the data is successfully modified | Audit log, send alert after data changes |
| ✅ **DDL Trigger** | **Schema level (DDL)** – CREATE, DROP, ALTER | **After** schema change commands are run | Prevent drop, log table creation or schema change |

**✅ What is a Cursor (Super Simple)?**

**A cursor is like a loop in SQL.**

**💡 Imagine a table has many rows — but you want to:  
👉 Do something for each row one by one, like print, check, or update.**

**SQL normally works with all rows at once (set-based),  
but a cursor lets you work row-by-row, like this:**

**✅ Situation:**

You have a table with multiple employees, and you want to:

👉 Add ₹5000 **only** to employees whose salary is **less than ₹30000** —  
But you want to **do it row by row**, and maybe apply **custom logic for each**.

**🔍 Why Cursor Helps:**

Normally in SQL, we do:

sql

CopyEdit

UPDATE Employees

SET Salary = Salary + 5000

WHERE Salary < 30000;

✅ This is **fast** but applies to **all rows at once**.

But if you want to:

* Track each update
* Log which rows were updated
* Add extra logic (like only if employee ID is odd, or send email, etc.)
* Handle **complex logic** per row

👉 Then **cursor is best** ✅

**🔄 Cursor – Row-by-Row Update Example**

**Step-by-step with Cursor:**

1. Start cursor – it reads first row (ID = 1, Salary = 25000)
2. Checks: Is salary < 30000? ✅  
   ➤ Add ₹5000 → New salary = 30000  
   ➤ Log this change
3. Go to next row (ID = 2, Salary = 30000)  
   ➤ Salary is not less than 30000 ❌ → Skip
4. Go to next row (ID = 3, Salary = 35000)  
   ➤ Also skip
5. Finish.

**🧠 Advantage of Cursor**

| **You Can...** | **Because Cursor Goes Row-by-Row ✅** |
| --- | --- |
| Add condition per row | ✅ Yes |
| Log or print row info | ✅ Yes |
| Handle complex business logic | ✅ Yes |
| Send emails or call stored procedures | ✅ Yes |
| Use loops and variables | ✅ Yes |

**🔧 Final Thoughts**

🧱 **Cursors = Useful when logic is too complex for plain SQL**  
But ⚠️ don't use them for big data unless truly needed — they are **slower**.

**✅ What is a Transaction?**

A **Transaction** is a group of **SQL statements** that must all succeed together — or **fail together**.

🧠 Think of it like:

“All-or-Nothing” ✅

**🎯 Real-Life Example:**

💳 Suppose you are transferring ₹1000 from Account A to Account B:

1. **Subtract** ₹1000 from A
2. **Add** ₹1000 to B

👉 Both should succeed ✅  
👉 If one fails, **none should happen** ❌

That’s a **Transaction**.

**🔐 ACID Properties of Transactions**

ACID = Four key rules that ensure data quality and safety.

| **Property** | **What It Means** |
| --- | --- |
| **A – Atomicity** | All steps happen together, or none happen (All-or-nothing). |
| **C – Consistency** | Data remains valid before and after the transaction. |
| **I – Isolation** | Transactions don’t affect each other, even if run at the same time. |
| **D – Durability** | Once committed, data is saved permanently—even if power goes out. |

**💡 Simple Analogy: Online Order**

| **Step** | **Matches ACID Rule** |
| --- | --- |
| Add to cart, place order | **Atomicity** |
| Only buy if item is in stock | **Consistency** |
| Your order doesn’t clash with others’ orders | **Isolation** |
| You get order confirmation—even if server shuts down | **Durability** |

**🧠 How Do You Still Get the Confirmation Even If the Server Shuts Down?**

This happens because of the **"D" in ACID**:

**✅ Durability = Once data is committed, it’s permanently saved — even if power goes off.**

**🔌 Real-Life Example:**

Let’s say you:

1. Place an order (SQL INSERT runs)
2. The database **commits** the transaction
3. Suddenly — 💥 Server power goes off

🔐 Still… your data is **already saved**!

**🔄 Scenario 1: Power OFF before COMMIT**

You ran INSERT or UPDATE… but didn’t run COMMIT yet.

**🧨 What Happens?**

* The change is in **temporary memory (RAM)**.
* It’s part of an **open transaction**.
* If **power goes off** 💥 before COMMIT:  
  ✅ **Nothing will be saved.**  
  🔄 Database **automatically rolls back** when it restarts.

💡 **No partial or half data will be saved** — this protects data integrity.

**🔐 Scenario 2: Power OFF after COMMIT**

You ran your query and did a COMMIT.

**🔒 What Happens?**

* The data was **written to transaction logs** first (Write-Ahead Logging).
* Then the COMMIT **flushes it to disk**.
* Even if **power goes off after that**, data is **safe** ✅

💡 That’s what we call the **Durability** property of **ACID**.

**🔧 How This Is Achieved Internally?**

| **Step** | **Stored Where** | **Volatile?** |
| --- | --- | --- |
| Query Runs (INSERT, etc) | Buffer cache (RAM) | ✅ Yes |
| Transaction log written | Disk | ❌ No |
| COMMIT issued | Forces log flush to disk | ❌ No |
| Server shuts down | On restart, logs are replayed or rolled back automatically |  |

**🔐 Safety Mechanisms Used:**

* ✅ **Write-Ahead Logging (WAL)**
* ✅ **Transaction Log Files**
* ✅ **Auto-recovery / rollback system**
* ✅ **Disk flush on commit**

**What is Indexing in SQL?**

Indexing is like a **book’s table of contents** — it helps SQL **find data faster** without scanning the whole table.

**🧠 Imagine This:**

You have a table with **10 lakh rows** (10,00,000).  
You want to **find a customer with ID = 1056**.

Without index:

* SQL reads **each row one-by-one** (called **Full Table Scan**). 😵

With index:

* SQL **jumps directly** to row where ID = 1056 (just like a page number!). ⚡