**✅ What is Database Normalization?**

**Database Normalization** is a process in relational database design used to:

* Organize data efficiently.
* Eliminate data redundancy (repeating data).
* Ensure data integrity.
* Simplify database maintenance.

It involves dividing large, complex tables into smaller, related tables and defining relationships between them using **keys**.

**Definition:**

**"Normalization is a systematic process of organizing data in a relational database to minimize data redundancy, eliminate undesirable data dependencies, and ensure data integrity."**

**Why is Normalization Needed?**

Without normalization:

* Data can be duplicated.
* Inconsistent data can exist (data anomalies).
* Insertion, deletion, and update operations can cause errors.

With normalization:

* Each piece of data is stored only once.
* Relationships between data are clearly defined.
* The database becomes easier to maintain and scale.

**Advantages of Normalization**

**1. Elimination of Data Redundancy**

* **What it means:** Same data is not stored in multiple places.
* **Why it matters:** Saves storage and prevents update anomalies.
* **Example:** Instead of storing a department name in every employee record, store it once in a separate department table.

**2. Improved Data Integrity and Consistency**

* **What it means:** Ensures accuracy and consistency of data throughout the database.
* **Why it matters:** When data is updated, it's updated in one place, so there's no conflicting information.
* **Example:** If a customer’s phone number changes, you only need to update it in one table.

**3. Efficient Data Update, Insert, and Delete Operations**

* **What it means:** Reduces anomalies in these operations.
* **Why it matters:**
  + **Update anomaly:** Same data needs to be updated in multiple places.
  + **Insert anomaly:** Can’t insert data due to missing related data.
  + **Delete anomaly:** Deleting data causes unintended loss of other data.
* **Example:** In a normalized schema, deleting an employee doesn't accidentally remove a department.

**4. Better Database Design**

* **What it means:** The database is modular and easier to understand.
* **Why it matters:** Improves scalability and makes maintenance easier.
* **Example:** You can change a part of the schema (like address format) without affecting unrelated parts.

**5. Scalability and Flexibility**

* **What it means:** Easier to modify the schema and expand functionality.
* **Why it matters:** Normalized databases adapt better to changes in requirements.
* **Example:** Adding new tables or attributes is more structured.

**🔶 Disadvantages of Normalization**

**1. Complex Queries and Joins**

* **What it means:** Since data is spread across multiple tables, queries involve complex joins.
* **Why it matters:** Increases query writing difficulty and can impact performance.
* **Example:** To get employee and department data, you need to join the Employee and Department tables.

**2. Performance Overhead**

* **What it means:** Excessive joins slow down the system, especially for read-heavy applications.
* **Why it matters:** Real-time systems (like e-commerce sites) may suffer delays.
* **Example:** Joining 5+ tables in a normalized schema can be slower than querying a single denormalized table.

**3. Initial Design Complexity**

* **What it means:** Requires in-depth knowledge of data and dependencies.
* **Why it matters:** Takes more time and skill to design a normalized schema.
* **Example:** Identifying all functional dependencies for 3NF or BCNF takes effort.

**4. Reduced Query Performance for OLAP**

* **What it means:** For analytical queries (e.g., reports), normalized structure can be inefficient.
* **Why it matters:** Data warehousing systems often prefer denormalization.
* **Example:** Running a monthly sales report may require joins across several normalized tables.

**5. Maintenance Complexity in Some Use-Cases**

* **What it means:** Changes in schema (like splitting or merging tables) may have wider ripple effects.
* **Why it matters:** Could require rewriting application queries or stored procedures.
* **Example:** Adding a new column that is shared across multiple tables might require schema refactoring.

**Balanced Perspective**

In practice:

* **Online Transaction Processing (OLTP)** systems (e.g., banking, order management) **prefer normalization** for integrity.
* **Online Analytical Processing (OLAP)** systems (e.g., dashboards, BI tools) **prefer denormalization** for speed.

**What is Functional Dependency?**

**A functional dependency (FD)** is a **relationship between two attributes (or sets of attributes)** in a relation (table), where **one attribute uniquely determines another**.

**Formal Definition:**

In a relation **R**, an attribute **Y** is **functionally dependent** on attribute **X** (written as **X → Y**) **if for every unique value of X, there is exactly one corresponding value of Y**.

**Example:**

Consider this **Employee** table:

| **EmpID** | **EmpName** | **Dept** |
| --- | --- | --- |
| 101 | Alice | HR |
| 102 | Bob | Finance |
| 103 | Alice | HR |

* **EmpID → EmpName**:  
  ✔ True, because each EmpID is unique and gives one EmpName.
* **EmpName → Dept**:  
  ❌ Not always true, because "Alice" could work in more than one department or multiple Alices could exist.

**Key Concepts in Functional Dependencies:**

| **Concept** | **Description** | **Example** |
| --- | --- | --- |
| **Determinant** | Left side of the FD (attribute that determines) | EmpID in EmpID → EmpName |
| **Dependent** | Right side of the FD (attribute being determined) | EmpName in EmpID → EmpName |
| **Trivial FD** | If Y ⊆ X in X → Y | EmpID → EmpID |
| **Non-trivial FD** | Y is not a subset of X | EmpID → EmpName |
| **Full Functional Dependency** | Y is fully dependent on X, and not on a part of X | (EmpID, DeptID) → Salary |
| **Partial Dependency** | Y depends on part of a composite key | StuID → DeptName (if composite key is StuID+CourseID) |
| **Transitive Dependency** | X → Y and Y → Z, then X → Z | EmpID → DeptID → DeptName |

**Why Functional Dependency is Important?**

Functional dependencies help:

* Identify **candidate keys**.
* Guide **normalization** (from 1NF to BCNF and beyond).
* Prevent **data anomalies**.
* Design **efficient and accurate** relational schemas.

**📌 Real-Life Analogy**

Imagine your **student roll number** determines your **name and department**:

* **RollNo → Name, Dept**
* For every roll number, we know exactly which student and department it maps to.

That’s a **functional dependency**!

**Normalization Process – The Normal Forms (NF)**

Normalization is carried out through a series of **normal forms** – each one building on the previous.

**1. 1NF (First Normal Form):**

* Ensures each column contains atomic (indivisible) values.
* No repeating groups or array.

**2. 2NF (Second Normal Form):**

* Must be in 1NF.
* Removes **partial dependencies** (non-key attributes dependent on part of a composite key).

**3. 3NF (Third Normal Form):**

* Must be in 2NF.
* Removes **transitive dependencies** (non-key attribute depending on another non-key attribute).

**4. BCNF (Boyce-Codd Normal Form):**

* Stronger version of 3NF.
* Every determinant must be a candidate key.

**5. 4NF (Fourth Normal Form):**

* Removes **multi-valued dependencies**.

**6. 5NF (Fifth Normal Form):**

* Deals with **join dependencies** and ensures lossless decomposition.

**Question**

**Given Relation:**

**Student(StudentID, StudentName, DeptName, HODName)**

**Given Functional Dependencies (FDs):**

1. **StudentID → StudentName, DeptName**
2. **DeptName → HODName**

**Goal: Normalize sTUDENT into 1NF → 2NF → 3NF**

**Step 1: First Normal Form (1NF)**

* **1NF requires that all attributes contain atomic values only.**
* **Assume the relation is already in 1NF (no multivalued or composite attributes).**

**Step 2: Second Normal Form (2NF)**

* **2NF = 1NF + No Partial Dependency on any candidate key.**
* **Candidate Key = StudentID (since it uniquely identifies students)**

**From FD 1:**

* **StudentID → StudentName, DeptName**

**From FD 2:**

* **DeptName → HODName  
  This introduces transitive dependency (not partial), so we continue to 3NF.**

**No partial dependencies → already in 2NF.**

**Step 3: Third Normal Form (3NF)**

* **3NF = 2NF + No Transitive Dependency on candidate key**

**Transitive dependency:  
StudentID → DeptName → HODName  
So we break the relation.**

**Decomposed Relations in 3NF:**

1. **Student(StudentID, StudentName, DeptName)**
   * **PK: StudentID**
   * **FD: StudentID → StudentName, DeptName**
2. **Department(DeptName, HODName)**
   * **PK: DeptName**
   * **FD: DeptName → HODName**

**Final Answer (3NF Relations):**

**Student(StudentID, StudentName, DeptName)**

**Department(DeptName, HODName)**