**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.

**Key concepts of Normalzation**

**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).
* **functional dependency X→A , X is a superkey.**
* **Or functional dependency X→A, A is a prime attribute.**

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

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

A relation is in **BCNF (Boyce-Codd Normal Form)** if and only if:

**For every functional dependency X→A , X must be a superkey.**

**Understanding BCNF:**

* A **superkey** is a set of attributes that uniquely determines all attributes in a relation.
* If there exists a functional dependency X→ where X is **not a superkey**, then the relation **violates BCNF** and needs decomposition

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

**Definition:**  
A relation is in **Fourth Normal Form (4NF)** if it is in **Boyce-Codd Normal Form (BCNF)** and **does not have multi-valued dependencies** (MVDs), except for trivial ones.

#### ****Multi-Valued Dependency (MVD)****

A multi-valued dependency exists when one attribute in a relation uniquely determines multiple values of another attribute, independent of other attributes.

* Removes **multi-valued dependencies**.

**Difference Between Functional Dependency (FD) and Multi-Valued Dependency (MVD)**

* **Functional Dependency (FD)**:
  + **X → Y** means that if two tuples have the same value for **X**, they must have the same value for **Y**.
  + Example: **Emp\_ID → Emp\_Name** (Each employee ID corresponds to exactly one name.)
* **Multi-Valued Dependency (MVD)**:
  + **X →→ Y** means that if two tuples have the same value for **X**, then they should be able to pair with **all possible values of Y**, independent of other attributes.
  + Example: **Prof\_ID →→ Course\_Taught** means that a professor can teach multiple courses, and these courses are independent of the languages they speak.

**Why Does 4NF Require Removing MVDs?**

* If a table has **X →→ Y**, then **X → Y does not hold** (it is not a functional dependency).
* This causes **unnecessary redundancy** and leads to **update anomalies**.
* **4NF eliminates these MVDs** by decomposing the table into **two separate relations**.
* **X → Y (Single Arrow)**: Functional Dependency (FD)
* **X →→ Y (Double Arrow)**: Multi-Valued Dependency (MVD)
* **MVD means that Y values are independent of Z values when grouped by X.**
* **4NF removes such MVDs to prevent redundancy and anomalies.**

**Example of Multi-Valued Dependency:**  
Consider a university database where a professor can teach multiple courses and speak multiple languages. These attributes are independent of each other.

| **ProID** | **Course\_Taught** | **Language\_Known** |
| --- | --- | --- |
| P1 | DBMS | English |
| P1 | DBMS | French |
| P1 | OS | English |
| P1 | OS | French |
| P2 | AI | English |
| P2 | AI | Hindi |
| P2 | ML | English |
| P2 | ML | Hindi |

Here, **Prof\_ID →→ Course\_Taught** and **Prof\_ID →→ Language\_Known** (independent multi-valued dependencies).  
This violates 4NF because **Course\_Taught and Language\_Known should be in separate relations**.

Even though the table is in BCNF, it **violates 4NF** because it contains **multi-valued dependencies (MVDs)**:

* **Prof\_ID →→ Course\_Taught**
* **Prof\_ID →→ Language\_Known**

These MVDs indicate that **Course\_Taught and Language\_Known are independent of each other** but are both dependent on **Prof\_ID**.  
4NF **does not allow non-trivial MVDs**, so we must decompose the table.

1**Professor-Course Relation**

| **Prof\_ID** | **Course\_Taught** |
| --- | --- |
| P1 | DBMS |
| P1 | OS |
| P2 | AI |
| P2 | ML |

2**Professor-Language Relation**

| **Prof\_ID** | **Language\_Known** |
| --- | --- |
| P1 | English |
| P1 | French |
| P2 | English |
| P2 | Hindi |

* Here, **Prof\_ID uniquely determines a set of languages**, but there are **no other independent attributes** in the table.
* Since **Language\_Known depends solely on Prof\_ID**, it behaves like an **FD (Prof\_ID → Language\_Known)** in this table.
* **There is no other independent attribute like Course\_Taught**, so there is **no MVD left** in this table.

Q Decompose below relation to 4th normal form

### ****Example: Student-Hobby-Skill Database****

#### ****Original Table (Before 4NF Decomposition)****

| **Student\_ID** | **Hobby** | **Skill** |
| --- | --- | --- |
| S1 | Chess | Java |
| S1 | Chess | Python |
| S1 | Music | Java |
| S1 | Music | Python |
| S2 | Painting | C++ |
| S2 | Painting | SQL |
| S2 | Dancing | C++ |
| S2 | Dancing | SQL |

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

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

The **Fifth Normal Form (5NF)**, also called **Project-Join Normal Form (PJNF)**, ensures that a relation is decomposed into smaller relations to remove redundancy while preserving the lossless join property.

* A relation is in **5NF** if and only if **every join dependency in it is implied by the candidate keys**. This means that a relation should be decomposable into two or more smaller relations without introducing redundancy.
* **Why is 5NF Needed?**

Even after achieving **4NF**, some tables may still suffer from redundancy due to **join dependencies**, which involve **multivalued dependencies among three or more tables**. **5NF eliminates such redundancy** by breaking the table further into smaller relations.

**What is a Join Dependency?**

A **Join Dependency (JD)** is a constraint that ensures a relation (table) can be **reconstructed (losslessly)** by joining multiple **projections** (subsets of attributes) of that relation.

Formally,  
A **relation R** has a **join dependency**:

JD(R1​,R2​,...,Rn​)

if:

R=πR1(R)⋈πR2(R)⋈⋯⋈πRn(R)R

Where:

Where:

* πRi(R) is the **projection** of relation **R** on attribute set **Ri.**
* ⋈ is the **natural join** operator.
* The result of joining all the projections must exactly reconstruct the original relation **R** (i.e., **lossless join**).

This means that the original relation **R** can be **perfectly reconstructed** by joining its projections on subsets of attributes.

Example 1

Let’s say we have a table Course\_Offering:

| **Student** | **Course** | **Instructor** |
| --- | --- | --- |
| Alice | DBMS | Prof. X |
| Alice | AI | Prof. Y |
| Bob | DBMS | Prof. X |
| Bob | AI | Prof. Y |

This table has three projections:

1. πStudent,Course
2. πCourse,Instructor
3. πStudent,Instructor ​

We can reconstruct the original table by joining these three.

But if there are tuples that **combine attributes wrongly** during joining (spurious tuples), then the decomposition is **not valid** unless a **join dependency** ensures the reconstruction is exact.

**Step 1: Understanding the Table Structure**

**Supplier\_Part\_Project**

| **Supplier** | **Part** | **Project** |
| --- | --- | --- |
| S1 | P1 | ProjA |
| S1 | P2 | ProjA |
| S2 | P1 | ProjA |
| S2 | P2 | ProjA |
| S1 | P1 | ProjB |
| S2 | P1 | ProjB |

**Candidate Key: (Supplier, Part, Project)**

* This is the **only combination** that uniquely identifies each row in the table.
* However, there exists a **hidden dependency** in the data that the candidate key does not fully capture.

**Step 2: Identifying the Join Dependency**

🔹 **Observations in the Table:**

* The data shows that:
  + Each **supplier** provides **certain parts**.
  + Each **project** requires **certain parts**.
  + Each **supplier** is assigned to **certain projects**.

🔹 **Key Issue:**

* The relationships **Supplier → Part, Part → Project, and Supplier → Project exist separately**.
* However, **there is no direct dependency between Supplier and Project** through the candidate key.
* The table is **storing unnecessary combinations** that can be derived **by joining three independent relationships**.

🔹 **Join Dependency Exists:**

* If we split the table into **three smaller relations**, we can reconstruct the original table **without losing information**:

1**Supplier\_Part (Which supplier provides which part?)**

| **Supplier** | **Part** |
| --- | --- |
| S1 | P1 |
| S1 | P2 |
| S2 | P1 |
| S2 | P2 |

2**Part\_Project (Which parts are used in which projects?)**

| **Part** | **Project** |
| --- | --- |
| P1 | ProjA |
| P2 | ProjA |
| P1 | ProjB |

3**Supplier\_Project (Which suppliers work on which projects?)**

| **Supplier** | **Project** |
| --- | --- |
| S1 | ProjA |
| S2 | ProjA |
| S1 | ProjB |
| S2 | ProjB |

**Reconstructing the Original Table**

* If we take the **natural join** of these three tables on "Supplier", "Part", and "Project", we get back the original table:

**Step 1: Original Table (Before Decomposition)**

We have an **Employee\_Department\_Role** table:

| **Employee** | **Department** | **Role** |
| --- | --- | --- |
| **E1** | HR | **Manager** |
| **E1** | HR | **Recruiter** |
| **E2** | IT | **Developer** |
| **E2** | IT | **Tester** |
| **E3** | Finance | **Accountant** |

**Candidate Key** = (Employee, Department, Role)  
**This table correctly shows who is assigned which role in which department.**

**Step 2: Incorrect Decomposition into Three Tables**

Now, let’s decompose it into **three separate tables**:

**1️Employee\_Department (Which employees work in which departments?)**

| **Employee** | **Department** |
| --- | --- |
| **E1** | HR |
| **E2** | IT |
| **E3** | Finance |

**2️ Department\_Role (Which roles exist in which departments?)**

| **Department** | **Role** |
| --- | --- |
| **HR** | Manager |
| **HR** | Recruiter |
| **IT** | Developer |
| **IT** | Tester |
| **Finance** | Accountant |

**3️ Employee\_Role (Which roles do employees have?)**

| **Employee** | **Role** |
| --- | --- |
| **E1** | Manager |
| **E1** | Recruiter |
| **E2** | Developer |
| **E2** | Tester |
| **E3** | Accountant |

**Step 3: Reconstructing the Original Table (Incorrect Data)**

Now, let’s **reconstruct the original table** by joining these three decomposed tables:

* **Step 1:** Join **Employee\_Department** and **Department\_Role** to find all possible (Employee, Department, Role) combinations.
* **Step 2:** Join the result with **Employee\_Role**.

What happens here?

* **E1 is in HR (from Employee\_Department)**.
* **HR has two roles: Manager and Recruiter (from Department\_Role)**.
* **E1 has two roles: Manager and Recruiter (from Employee\_Role).**

💡 **So, when we join these tables, we assume that E1 can take any role that exists in HR, even if that wasn’t originally the case!**

**Step 4: The Incorrect Extra Rows (Spurious Tuples)**

After joining all three tables, we **incorrectly generate extra rows**:

| **Employee** | **Department** | **Role** |
| --- | --- | --- |
| **E1** | HR | Manager |
| **E1** | HR | Recruiter |
| ❌ **E1** | **HR** | ❌ **(Any extra role from HR if more exist)** |
| **E2** | IT | Developer |
| **E2** | IT | Tester |
| ❌ **E2** | **IT** | ❌ **(Any extra role from IT if more exist)** |

⚠️ **Issue:** The decomposition makes us **incorrectly assume that an employee in a department can take any role from that department**.

**Step 5: Why This Is Wrong**

* **In the original table, the employee’s role in a department was explicitly defined.**
* **After decomposition, we lost the direct (Employee, Department, Role) relationship.**
* **The new tables only tell us:**
  + Which employees are in which departments.
  + Which roles exist in each department.
  + Which roles employees have (without department context).
* **When we join them back, the system assumes that an employee can take any role in their department, even if they were assigned only specific roles originally.**

**Example 2**

**Scenario: Course Offerings and Instructors**

Consider a university database tracking which **instructors** are assigned to teach specific **courses** in various **departments**. The initial relation might be structured as follows:

**Relation: Course\_Instructor\_Assignment**

| **Course\_ID** | **Department\_ID** | **Instructor\_ID** |
| --- | --- | --- |
| CSE101 | CSE | I1001 |
| CSE101 | CSE | I1002 |
| MAT202 | MAT | I1003 |
| PHY303 | PHY | I1004 |
| PHY303 | PHY | I1005 |

**Assumptions:**

1. An instructor can be associated with multiple departments.​
2. A course can be offered by multiple departments.​
3. An instructor can teach multiple courses across different departments.​

In this setup, the primary key is the combination of **(Course\_ID, Department\_ID, Instructor\_ID)**.​

**Identifying the Need for 5NF**

The relation above may contain redundancy due to the multi-faceted associations between courses, departments, and instructors. To eliminate redundancy and ensure that the data is represented efficiently without loss of information, we can decompose the relation into three separate tables:​

1. **Courses\_Departments**: Associates courses with the departments offering them.​
2. **Departments\_Instructors**: Associates departments with their instructors.​
3. **Courses\_Instructors**: Associates courses with instructors qualified to teach them.​

**Decomposed Relations**

1. **Courses\_Departments**

| **Course\_ID** | **Department\_ID** |
| --- | --- |
| CSE101 | CSE |
| MAT202 | MAT |
| PHY303 | PHY |

1. **Departments\_Instructors**

| **Department\_ID** | **Instructor\_ID** |
| --- | --- |
| CSE | I1001 |
| CSE | I1002 |
| MAT | I1003 |
| PHY | I1004 |
| PHY | I1005 |

1. **Courses\_Instructors**

| **Course\_ID** | **Instructor\_ID** |
| --- | --- |
| CSE101 | I1001 |
| CSE101 | I1002 |
| MAT202 | I1003 |
| PHY303 | I1004 |
| PHY303 | I1005 |

**Ensuring Lossless Join and Dependency Preservation**

By decomposing the original relation into these three tables, we ensure that:​

* **Lossless Join**: The original relation can be reconstructed by joining these tables on their common attributes.​
* **Dependency Preservation**: All functional and join dependencies are maintained without redundancy.

**DKNF (Domain-Key Normal Form)** is the **ultimate or ideal normal form** in relational database normalization. It was proposed by Ronald Fagin.

### 🔷 ****Definition of DKNF:****

A relation is in **Domain-Key Normal Form (DKNF)** if **all constraints** on the relation are a **consequence of domain constraints and key constraints only**.

* **Domain Constraints**: Restrict the permissible values of an attribute (e.g., age must be a positive integer).
* **Key Constraints**: Uniquely identify tuples in a relation (e.g., primary key, candidate key).

👉 If any **other types of constraints** (e.g., general constraints or business rules) exist, the relation **is not in DKNF**.

### 🔷 ****Motivation for DKNF:****

Other normal forms (1NF through 5NF) deal mainly with **redundancy** and **anomalies** caused by **functional, multivalued, or join dependencies**. However, some **semantic or business rules** cannot be captured through these dependencies. DKNF ensures that **all such constraints** are captured **structurally**, not procedurally (i.e., they don’t need to be enforced through application code or triggers).

### 🔷 ****Example of a Relation Not in DKNF:****

Let's consider a relation Course\_Enrollment:

| **StudentID** | **CourseID** | **Instructor** |
| --- | --- | --- |
| 101 | DBMS | Prof. Sharma |
| 102 | DBMS | Prof. Sharma |
| 103 | OS | Prof. Verma |

#### Constraints:

* **Key**: (StudentID, CourseID)
* **Domain**: StudentID is numeric, CourseID is a string, Instructor is a string.
* **Additional Constraint (Business Rule)**: "Each CourseID must be taught by only one Instructor."

This last constraint is **not** a domain or key constraint — it's a **business rule** (also called an **inference constraint**) that **cannot be enforced** by domain or key constraints alone.

Hence, this relation is **not in DKNF**.

### 🔷 ****How to Convert to DKNF:****

To achieve DKNF, decompose the relation to eliminate non-domain-key constraints.

**Step 1: Separate the business rule into another relation:**

Let’s create a new relation:

Course\_Instructor(CourseID, Instructor)

* Primary Key: CourseID
* This ensures each course is associated with only one instructor.

Now, update Course\_Enrollment:

Enrollment(StudentID, CourseID)

* Primary Key: (StudentID, CourseID)

Now all constraints are either:

* Key constraints (PKs), or
* Domain constraints (data types)

Now both relations are in **DKNF**.

### 🔷 ****Advantages of DKNF:****

* No redundancy.
* All constraints are structurally enforced (no need for external business rules or triggers).
* Eliminates update, insertion, and deletion anomalies **completely**.

### 🔷 ****Disadvantages of DKNF:****

* Often hard to achieve in practice.
* May lead to excessive decomposition.
* Complexity increases.
* Might reduce query performance due to too many joins.

**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.