**Day -3 --- DB Design: Normalization (1NF, 2NF, 3NF)**

**🧠 What is Normalization? (Super Easy)**

**🔧 It's like:**

Organizing your school notebooks 📚 — so that:

* ✂️ **No duplicate notes** are written again and again
* ✅ Everything is kept in the **right section**
* 🔗 Things are **connected properly** (e.g., notes link to subjects)

**📦 In a Database:**

Normalization means:

Keeping your data neat, clean, and in the right place.

**🎯 Main Goals of Normalization:**

| **Goal** | **Meaning (Simple)** |
| --- | --- |
| ❌ Avoid Repetition | Don't store same data again and again |
| ✅ Improve Accuracy | Keep data correct and updated in one place only |
| 🔗 Use Relationships | Link tables using **keys**, not copy-paste data |

**🔍 Real-Life Example:**

Imagine you are storing student info:

**❌ Without Normalization:**

| **Name** | **Course** | **Teacher** |
| --- | --- | --- |
| Rahul | Math | Mr. Ajay |
| Rahul | Science | Mr. Amit |

"Rahul" is repeated — not clean.

**✅ With Normalization:**

Break into 2 Tables:

**Students Table:**

| **StudentID** | **Name** |
| --- | --- |
| 1 | Rahul |

**Enrollments Table:**

| **StudentID** | **Course** | **Teacher** |
| --- | --- | --- |
| 1 | Math | Mr. Ajay |
| 1 | Science | Mr. Amit |

✔️ Now: No repeating name — just 1 Rahul in 1 place  
✔️ Other details linked using **StudentID**

**🤔 My Thought:**

“We already have constraints like UNIQUE, NOT NULL, and data types to avoid bad or duplicate data… so **why do we need normalization**?”

Normalization is like **architect**: It designs the whole building smartly so you don’t even need to repeat things.

**🎒 Imagine This Real-Life Scenario:**

You’re a college admin, and you're keeping records of students in **one big notebook** like this:

| **Student Name** | **Course** | **Teacher** |
| --- | --- | --- |
| Rahul | Math | Mr. Ajay |
| Rahul | Science | Mr. Amit |
| Priya | Math | Mr. Ajay |
| Priya | Science | Mr. Amit |
| Rahul | Math | Mr. Ajay ← this is repeated! |

You write **Rahul**, **Math**, **Mr. Ajay** again and again.

**❌ Problem Here:**

Even if Rahul is same person, you write his name again and again = **repetition**.

Even if the Math teacher is same (Mr. Ajay), you write his name again and again = **more repetition**.

**✅ Now Imagine Smart Solution — Normalization:**

You create **separate notebooks** like this:

📘 **Students Table**

| **student\_id** | **name** |
| --- | --- |
| 1 | Rahul |
| 2 | Priya |

📗 **Courses Table**

| **course\_id** | **course\_name** |
| --- | --- |
| 101 | Math |
| 102 | Science |

📕 **Teachers Table**

| **teacher\_id** | **teacher\_name** |
| --- | --- |
| 501 | Mr. Ajay |
| 502 | Mr. Amit |

📙 **Enrollments Table**

| **student\_id** | **course\_id** | **teacher\_id** |
| --- | --- | --- |
| 1 | 101 | 501 |
| 1 | 102 | 502 |
| 2 | 101 | 501 |

**💡 See the Magic?**

* You wrote **Rahul only once** ✅
* You wrote **Mr. Ajay only once** ✅
* All other details are linked by ID numbers (relationships) ✅
* No repetition — this is **normalization**!

**🧠 Final Understanding:**

* Constraints (like UNIQUE, NOT NULL) are like **police inside the room**: They watch data but only in one place.
* Normalization is like **architect**: It designs the whole building smartly so you don’t even need to repeat things.

**🔴 Without Normalization:**

**✅ PROBLEMS YOU MUST CONTROL MANUALLY (Hardcoded):**

1. **Duplicates**:  
   You have to **manually** check if a name or course or teacher is already in the table before inserting — hard work.
2. **Redundant Data = Bigger Table Size**:  
   If Rahul is in 3 courses, you write "Rahul" 3 times. You waste space.
3. **Update Nightmare (Update Anomaly)**:  
   If teacher “Mr. Ajay” changes name to “Mr. Ajay Kumar”, you have to update every row where his name appears.  
   ❌ Miss even one row? → **Wrong data stays.**
4. **Delete Danger (Delete Anomaly)**:  
   If you delete Rahul’s last course row, his name and info are gone too — forever.
5. **Data Integrity Issues**:  
   You can accidentally put course "Maath" instead of "Math" — no control. Spelling mistakes happen.

**✅ With Normalization (No Hardcoding Needed):**

Let’s see how **Normalization makes life easy**:

| **Problem** | **Without Normalization** | **With Normalization** |
| --- | --- | --- |
| Duplicate Check | Manually check values | Automatically handled via **Primary Key + Foreign Key** |
| Data Reuse | Write again and again | Just refer via **IDs** |
| Update | Update everywhere | Update in 1 place only |
| Delete | Dangerous | Safe with **CASCADE / RESTRICT** options |
| Consistency | Prone to typos | Controlled by **foreign key rules** |
| Storage | High table size | Much smaller and cleaner |

**Normalization is like a robot that auto-manages my database smartly and safely.”** ✅

**🤖 What Does the "Normalization Robot" Do?**

* **Splits your data** into smaller smart tables.
* **Links them** using keys (like student\_id, course\_id).
* **Follows strict rules** (1NF, 2NF, 3NF...) to reduce duplication.
* **Prevents human errors** by enforcing structure.
* **Saves memory**, ensures consistency, and boosts performance.

**✅ Q: Do we define Normalization for each table?**

**No**, we don’t *explicitly* "define normalization" for each table like a command or keyword.

🔸 Instead, **normalization is a *design approach*** — a set of **rules** you follow **while creating your database schema** (tables + relationships).  
🔸 So, **you apply normalization principles to the entire database design** — and that affects **how each table is designed**.

**I got it 100% right!** ✅

Normalization is a design approach — a set of planning rules.  
We need to ensure we follow these normalization rules while creating the table schema.  
If we follow them correctly, then our database is considered to be normalized.

**🔍 Key Clarification:**

* **It’s not a tool or a command** in SQL.
* It’s a **principle-based design method** you apply **during planning**.
* When you **design tables** in such a way that:
  + There’s **no redundant data**,
  + **Relationships use keys properly**, and
  + Data is **stored logically and efficiently**,

**✅ What I Asked:**

“While creating the table schema, what rules should we follow to ensure normalization?  
How do we know which rule belongs to which normal form?  
How do we apply normalization rules step by step?”

**🔧 Step-by-Step: Applying Normalization During Table Design**

**✅ Step 1: Start with a Raw Table (Unnormalized)**

You may receive data like this:

| **StudentID** | **Name** | **Courses Enrolled** | **Dept** |
| --- | --- | --- | --- |
| 1 | Rahul | Math, Physics | Science |
| 2 | Priya | Chemistry | Science |
| 3 | Rahul | Math, Chemistry | Science |

❌ Problems: Repeating values, duplicate names, multiple values in a single column.

**✅ Step 2: Apply 1NF (First Normal Form)**

**Rule:** No repeating or multivalued columns.

🔧 Action:

* Split multivalued data into separate rows.

| **StudentID** | **Name** | **Course** | **Dept** |
| --- | --- | --- | --- |
| 1 | Rahul | Math | Science |
| 1 | Rahul | Physics | Science |
| 2 | Priya | Chemistry | Science |
| 3 | Rahul | Math | Science |
| 3 | Rahul | Chemistry | Science |

✅ Now each cell has **atomic (single) values**.

**✅ Step 3: Apply 2NF (Second Normal Form)**

**Rule:** Remove **partial dependencies** (only for tables with composite primary keys).

🔧 Action:

* Separate data that does **not depend on the full primary key**.
* In our case, Name and Dept depend only on StudentID, not on Course.

So, we split into two tables:

**🧾 Students Table:**

| **StudentID** | **Name** | **Dept** |
| --- | --- | --- |
| 1 | Rahul | Science |
| 2 | Priya | Science |
| 3 | Rahul | Science |

**📚 Enrollments Table:**

| **StudentID** | **Course** |
| --- | --- |
| 1 | Math |
| 1 | Physics |
| 2 | Chemistry |
| 3 | Math |
| 3 | Chemistry |

Now, no data depends partially on a key.

**✅ Step 4: Apply 3NF (Third Normal Form)**

**Rule:** Remove **transitive dependencies** (non-key column depends on another non-key column).

In our case:

* Dept is dependent on StudentID, so no issue.
* But if we had:

| **StudentID** | **Name** | **DeptCode** | **DeptName** |
| --- | --- | --- | --- |

Then DeptName depends on DeptCode, not StudentID.  
👉 We'd create a **Departments table** separately:

| **DeptCode** | **DeptName** |
| --- | --- |
| SCI | Science |

And reference DeptCode in the Students table.

**✅ Final Structure After Normalization**

1. **Students**
   * StudentID (PK)
   * Name
   * DeptCode (FK → Departments)
2. **Departments**
   * DeptCode (PK)
   * DeptName
3. **Enrollments**
   * StudentID (FK)
   * Course

**🔍 How to Know Which Rules Are from Normalization?**

| **Normal Form** | **Rule Summary** | **What to Look For** |
| --- | --- | --- |
| 1NF | Atomic columns | Any multivalue cells? |
| 2NF | No partial dependency | Composite keys? Any column not dependent on the full key? |
| 3NF | No transitive dependency | Any non-key column depending on another non-key column? |

✅ **Yes, Paresh — you are absolutely correct!** 💯  
If I follow **all the rules of normalization (1NF, 2NF, 3NF, etc.)** during your table design, then:

**✔️ My database is considered normalized to that level.**

For example:

* If you follow only 1NF rules → your table is in **First Normal Form**
* If you follow 1NF + 2NF rules → your table is in **Second Normal Form**
* If you follow 1NF + 2NF + 3NF rules → your table is in **Third Normal Form**

Let’s break down **1NF**, **2NF**, and **3NF** — the core concepts of **Database Normalization** — in the **most simple and detailed way** possible:

**🧱 1NF (First Normal Form)**

**📌 Rule:**

* No **repeating groups** or **arrays** in a single column. (Not lists, arrays, or multiple items packed into one column.)
* Each **column must hold atomic (indivisible)** values. One column = One value only. Not a list, not a group. Just one single item.
* Each **row must be unique**. (Use a **Primary Key** (like student\_id) to **ensure uniqueness** of each row.)

**❌ Bad Example (Not in 1NF):**

| **StudentID** | **Name** | **Courses** |
| --- | --- | --- |
| 1 | Rahul | Math, Science |
| 2 | Priya | English, History |

🔴 "Courses" column is storing multiple values → **violates 1NF**

**✅ Corrected (1NF):**

| **StudentID** | **Name** | **Course** |
| --- | --- | --- |
| 1 | Rahul | Math |
| 1 | Rahul | Science |
| 2 | Priya | English |
| 2 | Priya | History |

✔️ Now each field holds **only one value**, and each row is **unique** → this is **1NF**

**🧱 2NF (Second Normal Form)**

**📌 Rule:**

* Must be in **1NF**
* **No partial dependency** on part of the primary key  
  (Only applies if table has **composite primary key**)
* **If a column depends on only part of the primary key (not the full key), that’s a partial dependency — and 2NF doesn’t allow it.**

**❌ Bad Example (Not in 2NF):**

| **StudentID** | **Course** | **StudentName** |
| --- | --- | --- |
| 1 | Math | Rahul |
| 1 | Science | Rahul |
| 2 | English | Priya |

Composite key = (StudentID, Course)  
🔴 StudentName depends **only** on StudentID, not Course → **partial dependency** → violates 2NF

**✅ Corrected (2NF):**

**Split into two tables:**

**📘 Students**

| **StudentID** | **StudentName** |
| --- | --- |
| 1 | Rahul |
| 2 | Priya |

**📘 Enrollments**

| **StudentID** | **Course** |
| --- | --- |
| 1 | Math |
| 1 | Science |
| 2 | English |

✔️ Now, no attribute depends on part of composite key → this is **2NF**

**🧱 3NF (Third Normal Form)**

**📌 Rule:**

* Must be in **2NF**
* **No transitive dependency**  
  (i.e., non-key column should not depend on another non-key column)
* **💡 What is a *Transitive Dependency*?**

A **transitive dependency** happens when:

* A **non-key column** depends on **another non-key column**,  
  instead of depending directly on the **primary key**.
* This is **not allowed** in 3NF.

**❌ Bad Example (Not in 3NF):**

| **StudentID** | **Name** | **DepartmentID** | **DepartmentName** |
| --- | --- | --- | --- |
| 1 | Rahul | 10 | Science |
| 2 | Priya | 20 | Arts |

🔴 DepartmentName depends on DepartmentID, which is **not a key** → **transitive dependency** → violates 3NF

**✅ Corrected (3NF):**

**📘 Students**

| **StudentID** | **Name** | **DepartmentID** |
| --- | --- | --- |
| 1 | Rahul | 10 |
| 2 | Priya | 20 |

**📘 Departments**

| **DepartmentID** | **DepartmentName** |
| --- | --- |
| 10 | Science |
| 20 | Arts |

✔️ Now all non-key attributes depend **only on the key** → this is **3NF**

🗝️ **Primary Key = StudentID**

Now check dependencies:

* StudentName depends on StudentID ✅
* DepartmentID depends on StudentID ✅
* ❌ DepartmentName depends on DepartmentID,  
  **not directly on StudentID** → ❌ Transitive Dependency

So this table is **not in 3NF**.

**🏁 Summary Table:**

| **Normal Form** | **Key Rule** | **Fixes What?** |
| --- | --- | --- |
| 1NF | Atomic values, unique rows | No repeating values or arrays |
| 2NF | No partial dependency | Every column fully depends on primary key |
| 3NF | No transitive dependency | No column depends on non-key column |

Yes, Paresh — **absolutely correct!** ✅  
Using **Primary Key**, **Unique Key**, and **Composite Key** properly is **deeply connected** to the **normalization process** in database design. They are not just optional steps — they are **core pillars** of normalization.