**Normalization**

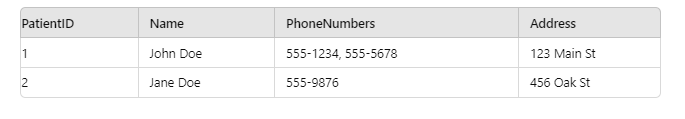
**1. First Normal Form (1NF)**

**Definition:** A table is in **1NF** if:

1. All the columns contain atomic (indivisible) values.
2. Each record is unique (no duplicate rows).

**Scenario:** Imagine a Patient table where we store patient information. A non-1NF table may have repeating groups of data in a single column.

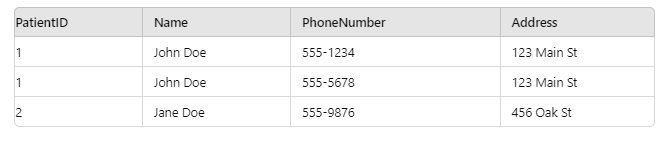
Non-1NF Table Example:



**Why is this not 1NF?**

* The PhoneNumbers column contains multiple values (e.g., "555-1234, 555-5678" for John Doe), which violates the atomic rule.

1NF-Compliant Table Example:



**What changed?**

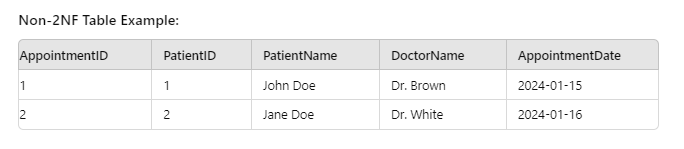
* Each PhoneNumber is now atomic (single value per cell).
* John's two phone numbers are now stored as two separate rows.

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

**Definition:** A table is in **2NF** if:

1. It is in 1NF.
2. All non-key attributes are fully dependent on the primary key (i.e., there are no partial dependencies).

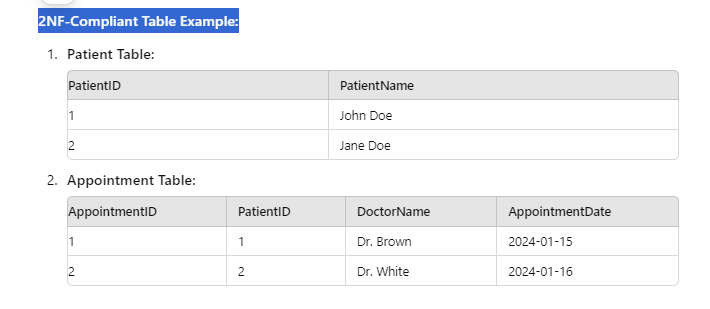
**Scenario:** Let’s consider a scenario where patients have appointments with doctors. Each appointment has a DoctorName, PatientID, and AppointmentDate. If we create a table that combines patient and appointment data, we could encounter partial dependency issues.



**Why is this not 2NF?**

* PatientName depends only on PatientID, but not on the entire composite key (AppointmentID and PatientID). This is a partial dependency, violating 2NF.

2NF-Compliant Table Example:



**What changed?**

* We separated patient information into its own Patient table, removing the partial dependency of PatientName on PatientID from the Appointment table.

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

**Definition:** A table is in **3NF** if:

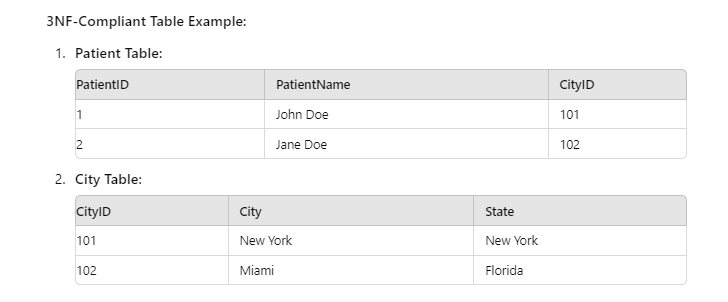
1. It is in 2NF.
2. There are no transitive dependencies (i.e., no non-key attribute depends on another non-key attribute).

**Scenario:** Let’s say each patient is associated with a particular city and the city is part of a state. If we store City and State in the Patient table, this introduces a transitive dependency because State depends on City.



**Why is this not 3NF?**

* The State attribute is dependent on City, not directly on PatientID. This creates a transitive dependency.



**What changed?**

* We separated City and State into a new City table. Now the Patient table only references CityID, removing the transitive dependency.

**Summary of Normal Forms:**

* **1NF:** Eliminate repeating groups and ensure atomic values.
* **2NF:** Eliminate partial dependencies (all columns must depend on the whole primary key).
* **3NF:** Eliminate transitive dependencies (non-key attributes must not depend on other non-key attributes).

**Denormalization.**

### ****Denormalization in SQL****

**Definition:**  
Denormalization is the process of intentionally adding redundancy into a database by combining tables or reintroducing duplicated data, which had been eliminated through normalization. The goal of denormalization is to improve read performance by reducing the number of joins needed to retrieve data.

While normalization focuses on minimizing redundancy and ensuring data integrity, denormalization trades these for improved query performance, especially in scenarios where joining multiple normalized tables can become slow.

### ****Why Denormalize?****

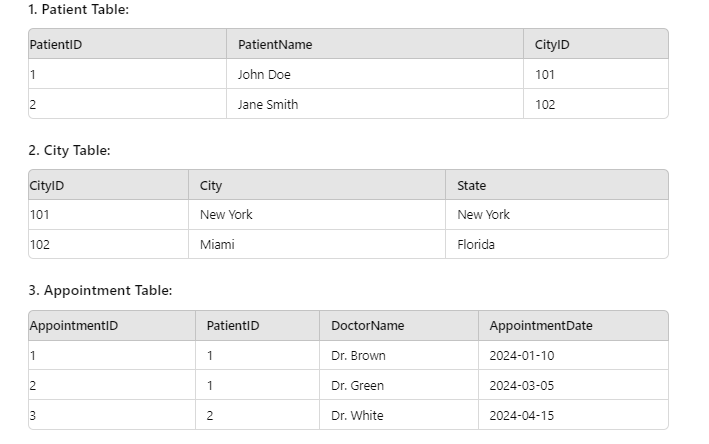
* **Improve Query Performance:** When databases have too many joins due to normalization (e.g., normalized to 3NF), the time taken to retrieve data can increase. By denormalizing, we can store frequently queried data together, reducing the number of joins required.
* **Simplify Complex Queries:** Joins across multiple tables can make queries harder to write, maintain, and optimize. Denormalization simplifies this by keeping related data together.
* **Use in Reporting Systems:** Denormalization is common in data warehouses, where the focus is on fast query performance rather than transaction-level consistency. Analytical queries often require data from multiple tables, and denormalized structures like star or snowflake schemas are often used.

### ****Example of Denormalization****

Let’s start with a **normalized** example and then show how denormalization could simplify it.

### ****Normalized Database (3NF Example)****

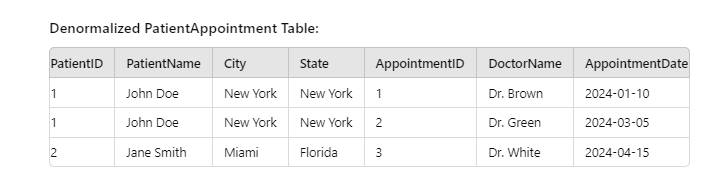
Suppose we have a medical database with patients and their appointments, normalized into **3NF**. The data is spread across multiple tables:

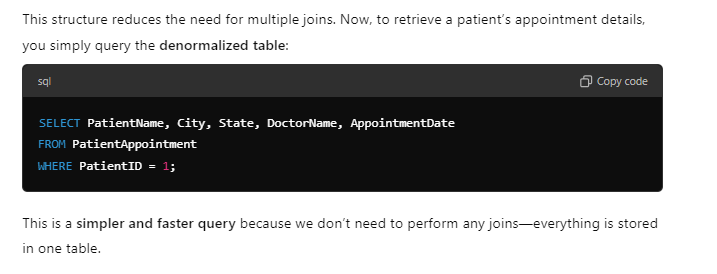


To retrieve patient details along with their appointments and city, we would need to join the Patient, City, and Appointment tables:

### ****Denormalized Database Example****

We can **denormalize** by combining the Patient, City, and Appointment data into a single table, which will allow faster query execution.





### ****Advantages of Denormalization****

1. **Faster Query Performance:** Fewer joins means faster queries, especially when querying large datasets.
2. **Simpler Queries:** Queries become easier to write and maintain since related data is pre-combined in a single table.
3. **Efficient Data Access:** Denormalization can be especially useful in read-heavy systems, where the priority is to retrieve data quickly.

### ****Disadvantages of Denormalization****

1. **Data Redundancy:** Duplicate data is reintroduced, which can increase storage requirements.
2. **Data Integrity Issues:** Since data is duplicated, changes need to be made in multiple places, increasing the chance of inconsistent or outdated data.
3. **More Complex Updates:** Updates become more complicated because changes to one record (e.g., a patient’s city) need to be propagated across multiple rows in the denormalized table.

### ****When to Use Denormalization?****

* **Analytical or Reporting Systems:** Systems focused on reading large datasets quickly, like data warehouses, often use denormalization to speed up queries.
* **Read-Heavy Applications:** If the system requires frequent reads with complex joins, denormalization can optimize performance.
* **Caching Data:** Denormalization can be used as a caching mechanism for frequently accessed data that would otherwise require complex joins.