◈ Class Note: Practical Database Normalization, Anomalies & Denormalization Using SQL

📃 Section 1: What is Normalization?

✅ Definition:

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity.

✅ Why Normalize?

To avoid data anomalies and ensure:

- Efficient storage
- Consistency
- Easy updates

⚠ Common Database Anomalies

Anomaly	Description	Example
Repetition	Same data repeated unnecessarily	Customer name stored in every order
Insertion	Can't insert data unless other unrelated data is also present	Can't add a product category unless a product exists
Update	Have to update the same data in many places	Price change needs update in every row

Anomaly	Description	Example
Deletion	Deleting a record removes important information	Delete last order → lose customer info

---

🔷 Section 2: How Bad Table Design Causes Anomalies

Let's start with a badly designed table in SQL:

```
CREATE TABLE Orders (
  OrderID INT,
  CustomerName VARCHAR(100),
  CustomerEmail VARCHAR(100),
  ProductName VARCHAR(100),
  ProductPrice DECIMAL(10,2),
  OrderDate DATE
);
```

🚨 Example Insert (DML):

```
INSERT INTO Orders VALUES
(1, 'Alice', 'alice@example.com', 'Laptop', 1500, '2025-06-01'),
(2, 'Alice', 'alice@example.com', 'Mouse', 20, '2025-06-01');
```

🔁 Repetition Anomaly:

- Customer info is repeated across rows

🔄 Update Anomaly:

```
-- Change Alice's email everywhere
UPDATE Orders SET CustomerEmail = 'alice@new.com' WHERE CustomerName = 'Alice';
```

If you forget one row → inconsistency.

🚫 Insertion Anomaly:

You can't add a new customer unless they've placed an order.

🗑 Deletion Anomaly:

```
DELETE FROM Orders WHERE OrderID = 1;
```

If this was the only order from Alice, we lose her contact too.

💠 Section 3: The 3 Normal Forms (With Fixes)

1️⃣ First Normal Form (1NF)

- **Goal:** Make values atomic (no repeated or multi-valued fields)
- **Fix:** Separate repeating groups

❌ Before (violates 1NF):

```
ProductName: 'Laptop, Mouse'
```

✅ After (1NF):

Each row should have just one product:

```
OrderID | ProductName
1       | Laptop
1       | Mouse
```

2️⃣ Second Normal Form (2NF)

- **Goal:** Remove **partial dependencies**
- **Occurs** when table has a **composite primary key** and some columns depend only on **part** of the key

❌ Bad Design:

```sql
CREATE TABLE OrderDetails (
  OrderID INT,
  ProductID INT,
  ProductName VARCHAR(100), -- depends on ProductID only
  Quantity INT,
  PRIMARY KEY (OrderID, ProductID)
);
```

✅ Good Design (2NF achieved):

Split into:

```sql
CREATE TABLE Products (
  ProductID INT PRIMARY KEY,
  ProductName VARCHAR(100)
);

CREATE TABLE OrderDetails (
  OrderID INT,
  ProductID INT,
  Quantity INT,
  PRIMARY KEY (OrderID, ProductID),
  FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

3️⃣ Third Normal Form (3NF)

- **Goal:** Remove **transitive dependencies**
- A non-key column should not depend on another non-key column

❌ Violates 3NF:

```
CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  CustomerName VARCHAR(100),
  CountryID INT,
  CountryName VARCHAR(50)
);
```

Here, `CountryName` depends on `CountryID`, which is not the primary key.

✅ Fix:

```
CREATE TABLE Countries (
  CountryID INT PRIMARY KEY,
  CountryName VARCHAR(50)
);

CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  CustomerName VARCHAR(100),
  CountryID INT,
  FOREIGN KEY (CountryID) REFERENCES Countries(CountryID)
);
```

🔍 Section 4: Summary of Dependencies

🔸 Partial Dependency

> A non-key column depends on part of a composite primary key.

Example:

In `OrderDetails(OrderID, ProductID, ProductName)`, `ProductName` depends only on `ProductID`.

🔸 Transitive Dependency

> A non-key column depends on another non-key column.

Example:

In `Customers`, `CountryName` depends on `CountryID`, which itself depends on `CustomerID`.

🌀 Section 5: Denormalization

✅ What is Denormalization?

The process of **adding some redundancy** to improve read performance or reduce JOINs.

❓ Why Denormalize?

- Reduce complex joins
- Speed up reporting
- Reduce query logic on front-end

❌ Normalized Tables:

```
-- Customers
CustomerID | Name | CountryID

-- Countries
CountryID | Name

-- Orders
OrderID | CustomerID | Date

-- Products
ProductID | Name | Price

-- OrderDetails
OrderID | ProductID | Quantity
```

To get full order info, you'd need 4-5 JOINS!

✅ Denormalized Version for Reporting:

```
CREATE TABLE OrderReport (
  OrderID INT,
  CustomerName VARCHAR(100),
  CountryName VARCHAR(50),
  ProductName VARCHAR(100),
  Quantity INT,
  ProductPrice DECIMAL(10,2),
  TotalPrice AS (Quantity * ProductPrice)
);
```

Now a simple query:

```sql
SELECT CustomerName, ProductName, TotalPrice
FROM OrderReport
WHERE CountryName = 'Nigeria';
```

Fast and simple — but space-consuming and redundant.

📝 Section 6: Exercises

1. Normalize This Table to 3NF

OrderID	CustomerName	ProductName	ProductPrice	CountryName

- Identify partial and transitive dependencies
- Redesign into multiple tables with SQL `CREATE TABLE` and `INSERT`

---

2. Write SQL to Show Anomalies

- **Update Anomaly:** Change product price in multiple rows
- **Insertion Anomaly:** Try to insert a product category without a product
- **Deletion Anomaly:** Delete an order, lose customer

---

3. Denormalize

- Join `Orders`, `Customers`, `Products`, `Countries` into one flat `OrderReport` table
- Write a `SELECT` query from this flat table for reporting

📌 Final Thoughts

✅ Normalize When	⚠️ Denormalize When
System must be scalable, consistent	Read performance is more important
Many writes/updates	Reports need instant access
Multiple apps use same DB	Data is rarely changed
