UNIT THREE- Relational Database Design

- **CONTENTS:-**
- **Relational Model**: Basic concepts, Attributes and Domains, CODD's Rules.
-
- **Relational Integrity**: Domain, Referential Integrities, Enterprise Constraints.
-
- **Database Design**: Features of Good Relational Designs, Normalization, Atomic Domains and First Normal Form, Decomposition using Functional Dependencies, Algorithms for Decomposition, 2NF, 3NF, BCNF.

Relational Model

What is Relational Model?

Relational Model is a key concept in Database Management Systems (DBMS) that organizes data in a structured and efficient way. It represents data and their relationships using tables. Each table has multiple columns, each with a unique name. These tables are also called relations. This model simplifies data storage, retrieval, and management by using rows and columns.

It primarily uses SQL (Structured Query Language) to manage and query data stored in tables with predefined relationships. SQL is widely used because it offers a consistent and efficient way to interact with relational databases.

The Relational Model is a fundamental concept in Database Management Systems (DBMS) that organizes data into tables. The relational model is widely used because it simplifies database management and ensures data accuracy. Codd's Rules, introduced by Dr. Edgar F. Codd, define the principles a database must follow to qualify as a true relational database.

These rules ensure data consistency, integrity, and ease of access, making them essential for efficient database design and management.

Some of the most well-known Relational database include **MySQL, PostgreSQL, MariaDB, Microsoft SQL Server, and Oracle Database**.

Characteristics of the Relational Model

- **Data Representation:** Data is organized in tables (relations), with rows (tuples) representing records and columns (attributes) representing data fields.
- **Atomic Values:** Each attribute in a table contains atomic values, meaning no multi-valued or nested data is allowed in a single cell.
- **Unique Keys:** Every table has a primary key to uniquely identify each record, ensuring no duplicate rows.
- **Attribute Domain:** Each attribute has a defined domain, specifying the valid data types and constraints for the values it can hold.
- **Data Independence:** The model ensures logical and physical data independence, allowing changes in the database schema without affecting the application layer.

- # Terminologies OR KEY CONCEPTS IN RELATIONAL MODEL

- **Relations (Tables):** It is the basic structure in which data is stored. Each relation is made up of rows and columns.
  Example: The table above named **Student**, is a relation. It stores data about students using rows and columns.

- **Relational Schema:** Schema represents the structure of a relation.
  Example: Relational Schema of STUDENT relation can be represented as STUDENT(StudentID, Name, Age, Course).

- **Relational Instance:** The set of values present in a relationship at a particular instance of time is known as a relational instance as shown in Table .

- **Attribute:** Each relation is defined in terms of some properties, each of which is known as an attribute. or Each column shows an attribute of the data.
  Example: StudentID, **Name**, **Age**, and **Course** are the attributes in this table STUDENT.

- **The domain of an attribute:** The possible values an attribute can take in a relation is called its domain.
  Example: The domain of the **Age** column is valid ages like 21, 22, 23 etc.
  The domain of the **Course** column includes valid courses like "Computer Science," "Mathematics," and "Physics."

- **Tuple:** Each row of a relation is known as a tuple.
  Example: STUDENT relation has 4 tuples.

- **Cardinality:** Cardinality refers to the number of distinct values in a column compared to the total number of rows in a table.
  Example: The **Age** column has 3 distinct values: 21, 22 and 23.

- **Degree (Arity):** The degree of relation refers to total number of attribute a relation has. It is also known as Arity.
  Example: The degree of this table is **4** because it has 4 columns: **StudentID, Name, Age** and **Course**.

1. Attributes in the Relational Model

An attribute is a column in a relational table. It represents a property or characteristic of the entity represented by the relation. Each attribute has a name. It has an associated domain (i.e., the set of permissible values). Attributes are the headings of columns in a table.

Key Points:

An attribute must have a unique name within a table.
Attributes define the structure of the relation.

2. Domains in the Relational Model

Definition:

A domain is the set of allowable values that an attribute can take. Every attribute has a domain. Domains help ensure data validity and consistency.

Key Points:

A domain may include data type, range, and format.
Domains can be shared by multiple attributes across different relations.

Codd's 13 Rule

Codd's Twelve Rules of Relational Database

Codd rules were proposed by E.F. Codd which should be satisfied by the relational model. Codd's Rules are basically used to check whether DBMS has the quality to become Relational Database Management System (RDBMS).These rules set basic guidelines to ensure data is stored and managed in a clear, consistent, and reliable way. But, it is rare to find that any product has fulfilled all the rules of Codd.
They generally follow the 8-9 rules of Codd. E.F. Codd has proposed 13 rules which are popularly known as Codd's 12 rules. These rules are stated as follows:

✅ Rule 0: Foundation Rule

A system qualifies as a relational database management system if it uses its relational capabilities to manage the database.

- **Meaning:** If a DBMS claims to be relational, it must manage data using only relational methods.

- **Example:** SQL Server or MySQL uses tables, keys, and relational operations (like JOIN) — so they qualify.

✅ Rule 1: Information Rule

All data should be represented as values in tables.

- **Example:** Student names, roll numbers, and marks are stored as values in rows and columns of a table.

RollNo	Name	Marks
101	Alice	85
102	Bob	90

✅ Rule 2: Guaranteed Access Rule

Every data item must be accessible by table name, primary key, and column name.

- **Example:** You can access Bob's marks using:
  SELECT Marks FROM Students WHERE RollNo = 102;

✅ Rule 3: Systematic Treatment of Nulls

Null values should be used uniformly to represent missing or inapplicable information.

- **Example:** If Alice's marks are not available, we store it as NULL.

RollNo	Name	Marks
101	Alice	NULL

✅ Rule 4: Dynamic Online Catalog (Data Dictionary)

The database should have a catalog (metadata) that can be queried using SQL.

- **Example:** You can query table structure using:
- SELECT * FROM information_schema.tables;

✅ Rule 5: Comprehensive Data Sub-language Rule

The system must support at least one language that supports data definition, manipulation, constraints, and access.

- **Example:** SQL can create tables, insert data, define constraints:
- CREATE TABLE Students (...);

✅ Rule 6: View Updating Rule

All views that can be updated theoretically should be updatable by the system.

- **Example:** If you create a view:
- CREATE VIEW HighScorers AS
- SELECT * FROM Students WHERE Marks > 80;

You should be able to update it:

UPDATE HighScorers SET Marks = 95 WHERE RollNo = 102;

✅ Rule 7: High-level Insert, Update, Delete

You should be able to manipulate sets of data (not just one row at a time).

- **Example:** Update all students who scored less than 50:
- UPDATE Students SET Marks = Marks + 5 WHERE Marks < 50;

✅ Rule 8: Physical Data Independence

Changes in physical storage (e.g., file location) should not affect how data is accessed.

- **Example:** Moving a table's storage file shouldn't affect your SQL queries.

✅ Rule 9: Logical Data Independence

Changes in the logical schema (e.g., adding columns) should not break existing applications.

- **Example:** Adding a new column like Email shouldn't break queries using only Name and Marks.

✅ Rule 10: Integrity Independence

Integrity constraints (like primary key, foreign key) should be stored in the catalog and not in application code.

- **Example:** Defining a constraint in SQL, not in application code:
- ALTER TABLE Students ADD CONSTRAINT pk PRIMARY KEY (RollNo);

✅ Rule 11: Distribution Independence

The system should work regardless of data distribution (centralized or distributed).

- **Example:** You shouldn't need to change queries whether the database is on one server or spread across many.

✅ Rule 12: Non-subversion Rule

You shouldn't be able to bypass integrity rules using low-level access.

- **Example:** You can't insert data that violates constraints using low-level tools like file editors or APIs.

Summary Table

Rule No.	Name	Meaning (Short)
0	Foundation Rule	Must use relational methods only
1	Information Rule	Data stored in tables
2	Guaranteed Access	Access data by table, row, and column
3	Systematic Null Treatment	NULLs must be uniform
4	Dynamic Catalog	Metadata is accessible via SQL

Rule No.	Name	Meaning (Short)
5	Comprehensive Language	Must support SQL-like language
6	View Updating	Updatable views must be supported
7	Set-Level Operations	Support bulk operations
8	Physical Independence	Storage changes shouldn't affect use
9	Logical Independence	Schema changes shouldn't break programs
10	Integrity Independence	Constraints managed by DBMS
11	Distribution Independence	Location of data doesn't matter
12	Non-subversion	Can't bypass DB rules with low-level tools

- # Relational Integrity

Relational Integrity refers to a set of rules and constraints in a relational database that ensures the accuracy, consistency, and validity of the data. It defines rules that maintain the logical relationships between data.

These rules prevent invalid data from being inserted, relationships from being broken, and business rules from being violated.

There are mainly three types of relational integrity constraints:

1. **Domain Integrity**
2. **Referential Integrity**
3. **Enterprise (Business) Constraints**

---

1. Domain Integrity / eitity integrity

- Ensures that attributes (columns) in a relation (table) have valid values.
- Every column has a defined domain (the set of permissible values).
- DBMS enforces that values inserted/updated in a column must belong to its domain.

The **domain** defines:

- Data type (e.g., integer, varchar, date)
- Range of values (e.g., age must be between 18 and 60)
- Format or pattern (e.g., email must match *@*.*)
- Specific set of values (e.g., gender = {Male, Female, Other})

Examples:

- Age must be an integer and greater than 0.
- Email must follow a valid email format.
- Gender can only be Male, Female, or Other.

Implementation in SQL:

```
CREATE TABLE Employee (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Age INT CHECK (Age > 18),
  Gender VARCHAR(10) CHECK (Gender IN ('Male','Female','Other'))
);
```

---

2. Referential Integrity

Referential integrity ensures that a foreign key value in one relation (child table) must either:

1. **Match an existing primary key value in another relation (parent table), OR**
2. **Be NULL (if allowed).**

- Ensures consistency among related tables.
- If a table (child) refers to another table (parent), the reference must be valid.
- Implemented using foreign keys.

Rules:

- A value in the foreign key column of child table must exist in the primary key column of parent table, or be NULL (if allowed).
- Prevents deletion/update in the parent if there are dependent rows in the child (unless handled with ON DELETE CASCADE or ON UPDATE CASCADE).

Example:

```
CREATE TABLE Department (
  DeptID INT PRIMARY KEY,
  DeptName VARCHAR(50)
);

CREATE TABLE Employee (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(50),
  DeptID INT,
  FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);
```

This ensures an employee cannot be assigned to a non-existent department.

---

3. Enterprise (Business) Constraints

Enterprise constraints in a Database Management System (DBMS) refer to high-level business rules that define the logical conditions and restrictions a database must follow to maintain data integrity and accurate modeling of real-world entities. These constraints go beyond the basic relational constraints (like primary keys and foreign keys) and are usually derived from the business logic of an organization.Implementation in SQL:

What are Enterprise Constraints in DBMS?

Enterprise constraints are:

- Also called **semantic constraints** or **business rules**.

- Rules or restrictions applied to data that reflect the **real-world operations** and **policies** of an organization.
- Used to **ensure data validity** according to the **specific needs of a business**.
- Often implemented using **triggers**, **assertions**, **procedures**, or **application-level logic**, since many DBMSs don't support them directly.

✅ Examples of Enterprise Constraints

Here are some typical examples of enterprise constraints:

Constraint Type	Example
Salary Rules	An employee's salary cannot exceed that of their manager.
Age Restrictions	An employee must be at least 18 years old.
Project Limits	A department cannot handle more than 5 projects simultaneously.
Loan Approval	A customer cannot take a second loan if the first is unpaid.

Difference Between Built-in and Enterprise Constraints

Aspect	Built-in Constraints	Enterprise Constraints
Definition	Standard constraints in the relational model	Business-specific rules
Examples	Primary key, foreign key, NOT NULL	Salary cap, project limits
Enforcement	Directly supported by DBMS	Often enforced via triggers, procedures
Flexibility	Fixed rules	Highly customizable

How Enterprise Constraints are Enforced

Since enterprise constraints are not always supported directly by DBMSs, they are typically implemented in the following ways:

1. **Triggers**
    - Automatically enforce rules before or after insert, update, or delete.
    - Example:
    - CREATE TRIGGER check_salary
    - BEFORE INSERT ON Employee
    - FOR EACH ROW
    - BEGIN
    - IF :NEW.salary > 100000 THEN
    - RAISE_APPLICATION_ERROR(-20001, 'Salary exceeds allowed maximum.');
    - END IF;
    - END;
2. **Stored Procedures**
    - Use logic encapsulated in procedures to manage inserts or updates with checks.
3. **Assertions** (in some DBMSs)
    - SQL standard supports them, but many DBMSs like MySQL do not.
    - Example:
    - CREATE ASSERTION age_check
    - CHECK (NOT EXISTS (
    - SELECT * FROM Employee WHERE Age < 18
    -));
4. **Application-level Checks**

- Business rules can also be enforced by the application code (Java, Python, etc.).

✅ **Summary Table**

Integrity Type	**Ensures…**
Domain Integrity	Values in a column are valid (correct type, format, range).
Referential Integrity	Relationships between tables remain consistent (foreign keys must match).
Enterprise Constraints	Business rules specific to the organization are enforced.

Database Design:

What is Database Design?

Database Design is the process of structuring data into well-organized tables (relations) so that:

- **Data is stored efficiently.**
- **Redundancy and inconsistency are minimized.**
- **Relationships among data are accurately represented.**
- **Data can be retrieved and updated easily without anomalies.**

It usually involves:

1. **Requirement Analysis** → understanding what data needs to be stored**.**
2. **Conceptual Design** → making an ER Diagram (entities, attributes, relationships).
3. **Logical Design →** converting ERD into relational schema (tables, keys, constraints).
4. **Normalization →** refining schema to remove redundancy and anomalies.
5. **Physical Design →** implementing tables, indexes, storage structures in DBMS.

Features of a Good Relational Design

A well-designed relational database should have the following qualities:

1. Minimal Redundancy

- Avoids duplicate storage of the same data.
- Prevents inconsistencies (e.g., same student's address stored in multiple places).
- Achieved through normalization.

---

2. No Insertion, Deletion, or Update Anomalies

- Insertion anomaly → Can't add a record because other unrelated data is missing.
- Deletion anomaly → Deleting a record unintentionally removes useful data.
- Update anomaly → Need to update the same data in multiple places.

A good design eliminates these anomalies.

---

3. Data Integrity

- Supports domain constraints, referential integrity, and enterprise rules.
- Ensures only valid, consistent, and accurate data is stored.

---

4. Clear Relationships

- All relationships (1:1, 1:M, M:N) are properly represented with primary keys and foreign keys.
- Prevents confusion and broken links between tables.

---

5. Flexibility and Scalability

- Easy to extend when requirements change (e.g., adding a new course for students).
- Schema should support growth in data volume without performance issues.

---

6. Good Performance

- Supports fast retrieval and updates using indexes and optimized schema.
- Avoids unnecessary joins and overly complex table structures.

---

7. Normalization with Balance

- At least in 3NF (Third Normal Form) or BCNF for minimal redundancy.
- But not over-normalized to the point where performance suffers due to too many joins.

---

8. Security and Access Control

- Sensitive data (like salaries, passwords) is properly controlled.
- Different users can access only what they are allowed to.

✅ **In Short:**
A good relational design =

- Efficient (minimal redundancy, optimized queries)
- Consistent (no anomalies, strong integrity)
- Flexible (can adapt to new requirements)
- Accurate (represents real-world entities and relationships correctly).

WHAT IS ANOMALIES? (Endsem)

Anomalies in SQL or DBMS refer to unexpected or undesirable behaviors that occur due to poor database design, especially when data is stored in an unnormalized form (not following normalization rules).

These anomalies can lead to inconsistencies, data redundancy, and data integrity issues.

Types of Anomalies in DBMS

There are **three main types** of anomalies:

1. **Insertion Anomaly**
2. **Update Anomaly**
3. **Deletion Anomaly**

Let's understand each with **real-time examples** using a simple **un-normalized table**.

Consider the Following Table: Student_Course

StudentID	StudentName	CourseID	CourseName	Instructor
101	Alice	CSE101	DBMS	Dr. Smith
102	Bob	CSE102	Operating Sys	Dr. Johnson
101	Alice	CSE102	Operating Sys	Dr. Johnson

This table violates normalization rules and can lead to anomalies.

---

1. Insertion Anomaly

Problem:

Occurs when you cannot insert data into the table without the presence of some other data.

Real TIME Example:

Suppose a **new course is added** ("CSE103 - Networks" taught by Dr. Lee), but **no student has enrolled yet**.

We **can't insert this course** because:

- The table requires a StudentID and StudentName, but we don't have any student yet for this course.

Consequence:

- We're **forced to insert NULLs** or **fake student data**, which violates data integrity.

---

2. Update Anomaly

Problem:

Occurs when **same data is repeated in multiple rows**, and updating it in one place but not in others causes inconsistencies.

Real Example:

Alice is enrolled in two courses. Her name is stored in multiple rows:

StudentID	**StudentName**	**...**
101	Alice	
101	Alice	

***Suppose Alice changes her name to "Alicia".**

Consequence:

- We need to update all rows where Alice appears.
- If we miss one, the database has **inconsistent data** (some rows say Alice, some Alicia).

---

3. Deletion Anomaly

Problem:

Occurs when **deleting some data unintentionally causes loss of additional useful data**.

Real Example:

Suppose Bob drops all his courses. We delete his record:

DELETE FROM Student_Course WHERE StudentID = 102;

Consequence:

- Bob's record is deleted, but also the course "CSE102" taught by Dr. Johnson may **no longer exist in the table** if no other student was enrolled in it.
- **We lose the course data**, even though the course might still be valid.

Solution: Normalization

To avoid these anomalies, we use **Normalization**, which involves dividing data into logical tables and establishing relationships.

✅ **Normalized Form Example:**

1. Student Table

StudentID	StudentName
101	Alice
102	Bob

2. Course Table

CourseID	CourseName	Instructor
CSE101	DBMS	Dr. Smith
CSE102	Operating Sys	Dr. Johnson

3. Enrollment Table

StudentID	CourseID
101	CSE101
102	CSE102

StudentID	CourseID
101	CSE102

✅ Benefits of Normalization

- No redundant data (reduces update anomalies)
- Easier to insert independent data (avoids insertion anomalies)
- Deleting one piece of data doesn't remove unrelated data (avoids deletion anomalies)

Summary Table

Anomaly Type	**Cause**	**Real-Life Example**	**Fix by Normalization**
Insertion Anomaly	Can't insert due to missing data	***Can't add a new course without a student***	Yes
Update Anomaly	Repeated data causes inconsistency	***Changing Alice's name in only one row***	Yes
Deletion Anomaly	Deletion causes loss of useful data	***Deleting Bob removes the course "CSE102" if no one else takes it***	Yes

Normalization in DBMS (Database Management System)

Normalization is a **step-by-step process** of organizing data in a database to remove redundancy and improve integrity. It uses **functional dependencies** to decide how to break tables.

- Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete, and update operations.
- In the relational model, there exist standard methods to quantify how efficient a databases is. These methods are called normal forms and there are algorithms to covert a given database into normal forms.
- Normalization generally involves splitting a table into multiple ones which must be linked each time a query is made requiring data from the split tables.

Normalization is the process of **organizing data** in a database to:

- Eliminate **redundancy** (repetitive data)
- Ensure **data integrity**
- Make the database more **efficient** and **flexible**

It involves dividing large tables into smaller, related tables and defining relationships between them.

🔷 Goals of Normalization

- Reduce data **redundancy**
- Avoid **anomalies** (insertion, update, deletion anomalies)
- Improve data **consistency**

🔵 Types (Forms) of Normalization:

There are several **normal forms (NF)**, each with specific rules. The most common are:

🔹 1NF (First Normal Form)

Rule:

- Each column should have **atomic (indivisible)** values
- Each row must be **unique**

🔸 **Example (Before 1NF - Not Normalized):**

StudentID	Name	Subjects
1	Alice	Math, Science
2	Bob	English, History

🔸 **After Applying 1NF:**

StudentID	Name	Subject
1	Alice	Math
1	Alice	Science
2	Bob	English
2	Bob	History

🔹 **2NF (Second Normal Form)**

Rule:

- Be in **1NF**
- **No partial dependency** (i.e., non-key attributes must depend on the **whole** primary key)

✅ Only applies to tables with **composite primary keys**

🔸 **Example (Before 2NF):**

StudentID	CourseID	StudentName	CourseName
1	101	Alice	Math

StudentID	CourseID	StudentName	CourseName
2	102	Bob	Science

- Here, StudentName depends only on StudentID, not on full key (StudentID, CourseID)

🔸 **After Applying 2NF:**

Students Table:

StudentID	StudentName
1	Alice
2	Bob

Courses Table:

CourseID	CourseName
101	Math
102	Science

Enrollments Table:

StudentID	CourseID
1	101
2	102

🔹 **3NF (Third Normal Form)**

Rule:

- Be in **2NF**
- No **transitive dependency** (i.e., non-key columns should not depend on other non-key columns)

🔸 **Example (Before 3NF):**

EmpID	EmpName	DeptID	DeptName
1	Alice	10	HR
2	Bob	20	IT

- DeptName depends on DeptID, not on EmpID

🔸 **After Applying 3NF:**

Employees Table:

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20

Departments Table:

DeptID	DeptName
10	HR
20	IT

Optional (Advanced Normal Forms):

🔹 **BCNF (Boyce-Codd Normal Form)**

- A stricter version of 3NF
- Every **determinant** must be a candidate key

🔹 **4NF (Fourth Normal Form)**

Rule:

Be in BCNF + No multi-valued dependencies.

Real-Time Example: Doctor's Specialties and Clinic Locations

Doctor_ID	Specialty	Clinic_Location
D01	Cardiology	Mumbai
D01	Neurology	Mumbai
D01	Cardiology	Delhi
D01	Neurology	Delhi

Note:- A doctor can have multiple specialties AND multiple clinic locations
→ Creates unnecessary combinations (Cartesian product)

AFTER (4NF):

DOCTOR_SPECIALTY

Doctor_ID	Specialty
D01	Cardiology
D01	Neurology

DOCTOR_LOCATION

Doctor_ID	Clinic_Location
D01	Mumbai
D01	Delhi

Each multi-valued dependency is in its own table → 4NF

5NF – Fifth Normal Form (Project-Join Normal Form)

Rule:

Be in 4NF + No join dependency that isn't implied by candidate keys.

✅ **Real-Time Example:** Product-Supplier-Country

Product	Supplier	Country
Laptop	Dell	USA
Laptop	Dell	Canada
Laptop	HP	USA
Laptop	HP	Canada

NOTE:- If each Product can be supplied by multiple suppliers and each supplier can operate in multiple countries, but not every combo is real → causes redundancy.

✅ **AFTER (5NF):**

PRODUCT_SUPPLIER

Product	Supplier
Laptop	Dell
Laptop	HP

SUPPLIER_COUNTRY

Supplier	Country
Dell	USA
Dell	Canada
HP	USA
HP	Canada

PRODUCT_COUNTRY (if needed based on actual deliveries)
→ Use only when needed to avoid unnecessary joins

✅ Breaks complex join dependencies → 5NF

Summary of Real-Life Scenarios

Normal Form	Real-Life Scenario
1NF	**Storing multiple phone numbers in one field**
2NF	**Student-course enrollment with extra student details**
3NF	**Employee-department relationship with department names**
BCNF	**Course assigning rooms, but course also determines room**
4NF	**Doctor with multiple specialties and clinic locations**
5NF	**Products, suppliers, and countries – complex combinations**

- No **multi-valued dependencies**
- Used in more complex relationships

✅ **Summary Table:**

Normal Form	Rules
1NF	Atomic values, no repeating groups
2NF	1NF + No partial dependency on composite key
3NF	2NF + No transitive dependencies
BCNF	3NF + Every determinant is a candidate key
4NF	BCNF + No multi-valued dependencies

What is Functional Dependency (FD)?

Functional Dependency (FD) is a **relationship between attributes** in a relational database. It expresses how **one attribute (or group of attributes) uniquely determines another attribute**.

Definition:

A functional dependency occurs when the value of one attribute (or a set of attributes) uniquely determines the value of another attribute. This relationship is denoted as:

X → Y

Here, **X** is the determinant, and **Y** is the dependent attribute. This means that for each unique value of X, there is precisely one corresponding value of Y.

If In a relation **R**, an attribute **A** is said to **functionally determine** attribute **B** (written as A → B) if:

For any two rows in R, if the values of A are the same, then the values of B must also be the same.

Use of Functional Dependency in Database Design:

- Helps in **normalization** (removing redundancy and anomalies)
- Identifies **candidate keys** and **prime attributes**
- Assists in decomposing tables into **3NF or BCNF**
- Improves **data integrity** and reduces **data redundancy**

Decomposition in Relational Model (DBMS)

Decomposition is the process of breaking a relation (table) into two or more smaller relations **without losing information** or introducing **redundancy or anomalies**. It's used primarily during **normalization** to ensure the database is well-structured.

✅ **Why Decomposition?**

To:

- Remove **redundancy**
- Avoid **update anomalies**
- Achieve **higher normal forms** (2NF, 3NF, BCNF, etc.)
- Ensure **data integrity**

Types of Decomposition

There are two main types of decomposition:

1. **Lossless (Lossless-Join) Decomposition**
2. **Lossy (Lossy-Join) Decomposition**

Let's break them down:

---

1. Lossless-Join Decomposition

A decomposition is **lossless** if you can **reconstruct the original relation** by joining the decomposed relations **without losing any data**.

Condition:

If a relation **R** is decomposed into **R1** and **R2**, the decomposition is lossless if:

R1 ∩ R2 → R1 or **R1 ∩ R2 → R2**

This means: the common attributes must be a **superkey** of at least one of the decomposed relations.

Example:

Let R(A, B, C) with FD: A → B

Decompose R into:

- R1(A, B)
- R2(A, C)

Since **A → B** (A is a key for R1), this decomposition is **lossless**.

---

2. Lossy-Join Decomposition

A decomposition is **lossy** if, when we join the decomposed relations, **we don't get back the exact original relation** — some information may be **lost or incorrect**.

This typically happens when **no functional dependency** connects the shared attributes.

Example:

Let R(A, B, C) with no functional dependency between attributes.

Decompose into:

- R1(A, B)
- R2(B, C)

Since B is not a key in either, the decomposition may be **lossy**.

✅ Desirable Properties of Decomposition

When decomposing a relation, we aim to satisfy:

1. **Lossless-Join**: No data is lost.
2. **Dependency Preservation**: All functional dependencies from the original relation are preserved in the decomposed relations.
3. **No redundancy**: Avoid unnecessary duplication.
4. **Higher Normal Forms**: The decomposed relations should ideally be in 2NF, 3NF, or BCNF.

Note: Sometimes, it's hard to achieve all of the above at once. For example, **BCNF** may require giving up **dependency preservation**.

Summary Table

Property	Description
Lossless-Join	Can reconstruct the original table
Dependency Preservation	All FDs are preserved
Normalization Goal	Reduce redundancy, anomalies
Used In	2NF, 3NF, BCNF decomposition

Example:-

Given Schema:

Student (RollNo, Branch_code, Marks_Obtained, Exam_Name, Total_Marks)

Let's identify functional dependencies (FDs) based on understanding:

Assumed Meaning of Attributes:

- **RollNo:** Unique student identifier
- **Branch_code:** Code of the student's branch (e.g., CS, EE)
- **Exam_Name**: Name of the exam (e.g., Midterm, Final)
- **Marks_Obtained:** Marks scored by student in the exam
- **Total_Marks**: Maximum marks possible in that exam

✅ **Step 1: Identify Functional Dependencies**

Likely FDs:

1. RollNo → Branch_code
   (Roll number uniquely determines the student's branch)
2. (RollNo, Exam_Name) → Marks_Obtained
   (Marks are based on which student and which exam)

3. Exam_Name → Total_Marks
   (Each exam has a fixed total marks value)

✅ **Step 2: Candidate Key(s)**

From the FDs:

- To uniquely identify a record (a student's marks in a specific exam), you need:
  - RollNo + Exam_Name

So, **Candidate Key** = {RollNo, Exam_Name}

✅ **Step 3: Check for 3NF**

🔹 **What is 3NF?**

A relation is in **Third Normal Form (3NF)** if:

1. It is in **Second Normal Form (2NF)**
2. **No transitive dependency** exists, i.e.,
   - Every non-prime attribute is **fully functionally dependent** on **every candidate key**
   - And **only** on candidate keys (i.e., no dependency on non-key attributes)

🔹 **Let's check each FD:**

1. RollNo → Branch_code
   - **Violation:**

     Branch_code is **dependent on part of the candidate key** (RollNo), not the full key (RollNo, Exam_Name)
   - This violates **2NF**, so not even in 2NF
2. (RollNo, Exam_Name) → Marks_Obtained
   - OK – fully functionally dependent on full candidate key
3. Exam_Name → Total_Marks

- **Violation**:

  Total_Marks is dependent on part of the candidate key (Exam_Name)

- Violates **2NF**

Conclusion:

The given schema is **not in 2NF or 3NF** because of **partial dependencies**.

✅ Step 4: Convert to 3NF

We decompose the table into smaller relations based on the FDs.

🔸 New Tables:

1. Student_Info

Student_Info (RollNo, Branch_code)

- RollNo → Branch_code

2. Exam_Info

Exam_Info (Exam_Name, Total_Marks)

- Exam_Name → Total_Marks

3. Student_Marks

Student_Marks (RollNo, Exam_Name, Marks_Obtained)

- (RollNo, Exam_Name) is the **composite key** here
- Marks_Obtained is fully dependent on this composite key

✅ Final 3NF Design:

Table Name	Attributes	Primary Key
Student_Info	RollNo, Branch_code	RollNo

Table Name	Attributes	Primary Key
Exam_Info	Exam_Name, Total_Marks	Exam_Name
Student_Marks	RollNo, Exam_Name, Marks_Obtained	(RollNo, Exam_Name)

✅ **Benefits of 3NF Conversion:**

- Eliminates **redundancy** (e.g., branch or total marks repeated)
- Prevents **update anomalies** (e.g., changing total marks in one place)
- Ensures **data consistency** and better structure

Thank you