UNIT IV
DATABASE DESIGN

DATABASE DESIGN

- **How to design good relational databases** — databases that store data efficiently, without redundancy, and without losing information.
- When we design a relational database, we decide **which attributes** (columns) go together in a **relation (table)**.
  But how do we know if our design is **good or bad**?
  That's what this chapter helps us understand — using **Functional Dependencies (FDs)** and **Normalization**.
- So far you've learned about:

✔ What a **relation** (table) is.

✔ How to define **attributes** (columns).

✔ How to create **queries** using SQL.

- But we didn't have a **formal method** to check whether our table design is correct.
  Now, we introduce **theory** that tells us:
- When a table design is **good**.
- When we need to **split (decompose)** tables to avoid problems.

Basics of Functional Dependencies and Normalization for Relational Databases

<u>Two Levels of “Goodness” in Design:</u>

Logical (Conceptual) Level

- How clearly users can understand what each table and column means.
- A good design here makes it easy to query and interpret data correctly.

Physical (Implementation) Level

- How the data is stored and updated efficiently in memory/disk.
- Mainly applies to **base tables** (real stored tables), not to views.

<u>Two Database Design Approaches:</u>

1. Bottom-Up (Design by Synthesis)

- Start by looking at relationships among individual **attributes**.
- Try to group them into tables.
- Not very practical — it’s hard to find all attribute relationships.

2. Top-Down (Design by Analysis)

- Start with larger natural groups (like forms, invoices, reports).
- Analyze them and **refine or decompose** step by step.
- **This is the more practical and preferred method.**

Basics of Functional Dependencies and Normalization for Relational Databases

Functional Dependencies (FD):

- A **functional dependency (FD)** is a relationship between two sets of attributes in a **relation (table)**.
  It shows how one set of attributes **uniquely determines** another.
- Formally,
  X → **Y** means:
  If two tuples (rows) have the same value for **X**, they must also have the same value for **Y**.

Example:

- Consider a relation STUDENT(RegNo, Name, Dept, Advisor)
- RegNo → Name, Dept, Advisor
  → Because **RegNo** uniquely identifies each student.

Example : Student Table

Functional Dependency:
RegNo → Name, Dept, Advisor

- Each student has a **unique registration number (RegNo)**.
- So, if two rows have the same **RegNo**, they must have the same **Name, Dept, and Advisor**.
- Hence, **RegNo** uniquely determines all the other attributes.

RegNo	Name	Dept	Advisor
101	Aruna	CSE	Dr. Rao
102	Ravi	ECE	Dr. Devi
103	Priya	CSE	Dr. Rao

Example : Composite Dependency

Functional Dependency:
(RollNo, Subject) → Marks

- The combination of RollNo and Subject **together** determines Marks.
- One attribute alone is not enough: RollNo alone doesn’t determine Marks (because one student has many subjects).

RollNo	Subject	Marks
1	DBMS	88
1	Java	90
2	DBMS	85

Normalization:

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

- Remove **redundancy (duplicate data)**,
- Avoid **anomalies (update, insert, delete issues)**,
- Ensure **data integrity**.

Informal Design Guidelines for Relation Schemas

- Before discussing the formal theory of relational database design, we discuss four informal guidelines that may be used as measures to **determine the quality of relation schema design**:

■ Making sure that the semantics of the attributes is clear in the schema

■ Reducing the redundant information in tuples

■ Reducing the NULL values in tuples

■ Disallowing the possibility of generating spurious tuples

1.Imparting Clear Semantics to Attributes in Relations

- this is a **very important foundational concept** in **database design.**
- **Semantics of Relation Schemas (Meaning in Relational Design)**
- Each relation should represent **one clear concept** (entity or relationship).
- Foreign keys link related entities.
- The **clearer** the meaning of a table, the **better** the database design.
- This principle is the **first informal guideline** for relational schema design.
- If the meaning of a table is **clear and unambiguous**, the database design is **good**

EMPLOYEE Relation:

Each row (tuple) in EMPLOYEE represents **one employee**.
Meaning (Semantics):
"This row represents an employee named Arun, whose SSN is 101, born on 12-12-1995, lives in Delhi, and works in department number 5."

- **Ename, Bdate, Address** → describe the employee
- **Ssn** → unique identifier (primary key)
- **Dnumber** → foreign key linking employee to DEPARTMENT

Clear and meaningful design
— all attributes describe the same real-world object: *an employee*.

Ename	Ssn	Bdate	Address	Dnumber
Arun	101	12-12-1995	Delhi	5

Informal Design Guideline

- **Guideline 1:**
  A good relation schema is one where the **meaning (semantics)** of tuples and attributes is **clear, simple, and easy to explain**.
- If you can say in one simple sentence what a row in the table represents (e.g., *"Each row in EMPLOYEE represents one employee"*), then your design is good.

Examples of Violating Guideline 1:
Example : EMP_DEPT Relation

Each **tuple (row)** represents an **employee**.
But in addition to employee information, it also contains **department details** (like Dname and Dmgr_ssn).

- **this a problem b**ecause this table is **mixing two different entities**:

- **Employee** (Ename, Ssn)
- **Department** (Dnumber, Dname, Dmgr_ssn)

This violates **Guideline 1**, since the table doesn't clearly represent a *single concept*.
It's unclear whether EMP_DEPT represents:

- An **employee**, or
- A **department**, or
- A **relationship between them**.

If a department has many employees, its data (like Dname, Dmgr_ssn) is **repeated** many times — one for each employee.
This repetition leads to **data redundancy** and **update problems**, which causes **update anomalies**.

Ssn	Ename	Dnumber	Dname	Dmgr_ssn
101	John	5	Research	333
102	Mary	5	Research	333
103	Tom	7	Sales	555

2.Redundant Information in Tuples and Update Anomalies

- The goal of **good database design** is to **avoid unnecessary repetition of data** and to **prevent problems during data updates** (insertion, deletion, modification).
- Redundant leads to wastage storage and can lead to **inconsistency.**

Understanding the Problem: Redundancy

Imagine two ways of designing your database:

Design A — Two Separate Tables:

- **EMPLOYEE**(Ssn, Ename, Dnumber)
- **DEPARTMENT**(Dnumber, Dname, Dmgr_ssn)

Here:

- Each department’s info (name, manager) is stored **once** in the DEPARTMENT table.
- Each employee’s info is stored **once** in the EMPLOYEE table, with **Dnumber** as a foreign key linking to DEPARTMENT.

Design B — One Combined Table (EMP_DEPT):

Here, department details (like **Dname** and **Dmgr_ssn**) are **repeated** for every employee in that department.

Ssn	Ename	Dnumber	Dname	Dmgr_ssn
101	John	5	Research	333
102	Mary	5	Research	333
103	Raj	7	Sales	555

What Problems Does Redundancy Cause?

- When you store such “joined” data (like EMP_DEPT), you face three types of **update anomalies**:
- **Insertion anomaly**
- **Deletion anomaly**
- **Modification anomaly**

Insertion Anomalies:

Insertion anomalies occur when you **can't insert new data easily** without adding unnecessary or incomplete information.

Example 1

Suppose you want to insert a **new employee** in EMP_DEPT.

To add: (104, Tina, 5, Research, 333)

- You must also fill in the department information (**Dname** and **Dmgr_ssn**) correctly and make sure it **matches** the existing values for department 5.

- If you make a typo (like writing *Reseach* or *Dmgr_ssn = 334*), the database becomes inconsistent.

In contrast, if you had separate tables (EMPLOYEE and DEPARTMENT), you only enter:

(104, Tina, 5)

- The department information already exists in the **DEPARTMENT** table — no duplication, no mismatch.

Example 2:

- You want to add a **new department** that currently has **no employees**.
- In EMP_DEPT, you can't do this easily — the table's primary key is **Ssn** (employee ID).
- You'd need to enter **NULL** values for employee attributes — which is not allowed for a primary key.
- So, you can't store a department unless it already has employees.

But if you had a **DEPARTMENT** table separately, you could simply insert:

(9, HR, 666)

even if there are no employees yet.

<u>Deletion Anomalies:</u>

A **deletion anomaly** happens when deleting one piece of information **accidentally removes other useful data**.

Example:

If you delete the employee **Raj (Ssn = 103)** from EMP_DEPT, and Raj was the **only employee in department 7**,
→ the information about **department 7 (Sales)** will be **lost entirely** from the database!

In the correct design (two tables):

- Deleting Raj only removes his record from **EMPLOYEE**.
- Department 7's info still remains safe in **DEPARTMENT**.

<u>Modification Anomalies:</u>

- A **modification anomaly** occurs when you change some information in one row but forget to update it everywhere else — leading to inconsistent data.

Example:

Suppose the **manager of department 5** changes from 333 to 777.

- In EMP_DEPT, you must update this value **for every employee** in department 5.
- If there are 50 employees in department 5 and you update only 49 of them, — now your database says department 5 has **two different managers!**

In the correct design (DEPARTMENT table):
You update **one row only**, and it's automatically consistent for all employees who reference department 5.

The Design Guideline (Guideline 2)

- **Guideline 2:**
  Design base relations so that there are **no insertion, deletion, or modification anomalies**.
- If anomalies are unavoidable, document them and handle them properly in update programs or triggers.

Summary:

Type of Anomaly	What It Means	Example Problem
Insertion	Can’t add data unless you add extra or duplicate info	Can’t add new dept without employee
Deletion	Deleting one thing removes something else important	Delete last employee → lose department
Modification	Must update same data in many rows	Change manager name → update all rows

Note:

Bad designs (like EMP_DEPT) may look convenient at first (all data in one place), but they lead to:

- Repetition of data (wasted space)
- Risk of inconsistency
- Difficulties in updating

So, good schema design:

- **Stores each fact only once**
- **Uses foreign keys** to connect related entities
- **Eliminates anomalies**

When Violations Are Sometimes Acceptable

- Sometimes, we **intentionally keep redundant data** (like EMP_DEPT) for **faster query performance** — such a table is called a **materialized view**.

In that case, we must ensure that:

- Whenever the base data (EMPLOYEE or DEPARTMENT) changes,

→ the redundant table is automatically updated by **triggers** or **stored procedures** to stay consistent.

Summary

Concept	Description
Redundancy	Repetition of same information in multiple places
Insertion Anomaly	Trouble adding new data because of dependency on other data
Deletion Anomaly	Losing important info when deleting related data
Modification Anomaly	Inconsistent data after partial updates
Goal	Avoid redundancy and anomalies using proper schema design
Solution	Separate entities into individual tables and connect them using foreign keys

NULL Values in Tuples

- Sometimes in a database, we design a relation (table) that includes **too many attributes** — some of which don't apply to every record.
- When that happens, we often leave those fields **empty or NULL**. But having too many **NULL values** in a table can lead to **storage waste, confusion, and errors in queries**.

What is a NULL Value?

A **NULL value** in a database means **“no value”** or **“unknown value.”** However, **NULL doesn’t always mean the same thing** — it can mean one of several things depending on the situation:

Meaning	Example
Value not applicable	A *Visa_status* column doesn’t apply to U.S. students (only for foreign students).
Value unknown	An employee’s *Date_of_birth* isn’t known yet.
Value exists but not recorded	The employee *Home_Phone_Number* exists but hasn’t been entered yet.

The Problem: “Fat Relations” and Too Many NULLs

-When a table has **many columns (attributes)**, it is called a **“fat relation.”**

For example:

- **Many NULLs** appear because not all employees get a *bonus*, *commission*, or *have a visa*.
- This wastes **storage space**.
- It also makes the **table confusing** — it’s not clear which NULLs mean “not applicable,” and which mean “unknown.”

Emp_ID	Name	Salary	Bonus	Commission	Visa_status	Office_number
101	John	60000	5000	NULL	NULL	220
102	Mary	55000	NULL	4000	NULL	NULL
103	Li	65000	6000	NULL	F1	NULL

Why NULL Values Cause Problems

(a) Storage Waste

- Every NULL still occupies some memory space.
- If many attributes are NULL for most rows, the database wastes storage unnecessarily.

(b) Query Confusion

- When you use **SQL queries** with NULLs, the results can be confusing because NULLs are treated differently in comparisons.

Example:

SELECT * FROM EMPLOYEE WHERE Bonus = 0;

This will **not show employees with NULL Bonus**, because:

- NULL ≠ 0
- NULL ≠ anything — not even another NULL!

So, you might think there are no employees with missing bonus info, but that's wrong — the query just ignores them.

Design Guideline (Guideline 3)

- As far as possible, **avoid placing attributes** in a base relation whose values may frequently be **NULL**.
- If NULLs are unavoidable, ensure they occur **only in rare, exceptional cases**, not for most of the tuples.

Problem	Caused By	Why It's Bad	Solution
Too many NULLs	Adding attributes that don't apply to all records	Wastes space and confuses meaning	Separate into smaller related tables
Ambiguous meaning	NULL can mean "not applicable," "unknown," or "missing"	Hard to interpret	Document or avoid multiple meanings
Query errors	NULLs ignored in JOINs or comparisons	Incorrect results	Use IS NULL / IS NOT NULL conditions
Aggregate function errors	SUM, COUNT skip NULL values	Wrong calculations	Handle NULLs explicitly with COALESCE or separate tables

Generation of Spurious Tuples

What are "Spurious Tuples"?

- **Spurious tuples** are *extra, incorrect rows* that appear when two relations are joined improperly.
- They represent **false information** — data that didn't exist in the original database but was created due to an **improper join**.
- These wrong results usually happen when relations are **not connected by proper primary key–foreign key pairs**, and we try to join them using some common attribute that isn't unique.

<u>Example: EMP_PROJ Relation</u>

Let's start from a correct relation:

EMP_PROJ

Here, every tuple tells us which **employee** works on which **project** and how many **hours** they work per week.

Ssn	Ename	Pnumber	Pname	Plocation	Hours
101	John	10	Alpha	Dallas	10
101	John	20	Beta	Houston	15
102	Mary	20	Beta	Houston	20

Now, a Bad Design: Decomposition into Two Relations

Suppose someone decides to "simplify" this table by splitting it into two smaller ones:

1. **EMP_LOCS(Ename, Plocation)**
→ means "Employee Ename works on *some* project at location Plocation."
2. **EMP_PROJ1(Ssn, Pname, Pnumber, Plocation, Hours)**
→ means "Employee with Social Security Number Ssn works for Hours per week on project Pname at location Plocation."

Now, a Bad Design: Decomposition into Two Relations

Suppose someone decides to “simplify” this table by splitting it into two smaller ones:

1. **EMP_LOCS(Ename, Plocation)**
   → means “Employee Ename works on *some* project at location Plocation.”
2. **EMP_PROJ1(Ssn, Pname, Pnumber, Plocation, Hours)**
   → means “Employee with Social Security Number Ssn works for Hours per week on project Pname at location Plocation.”

Ssn	Ename	Plocation
101	John	Dallas
101	John	Houston
102	Mary	Houston

Ssn	Pnumber	Pname	Hours
101	10	Alpha	10
101	20	Beta	15
102	20	Beta	20

The Problem

Now, if we try to **rebuild the original EMP_PROJ** by performing a **NATURAL JOIN** between EMP_LOCS and EMP_PROJ1 on the common attribute **Plocation**, we get something like this:

Result of JOIN:

The last tuple (marked) is **not real** —

Mary never worked on project Alpha in Dallas!

But it appeared because **Plocation** was not a key — multiple projects share the same location.

Ssn	Ename	Pnumber	Pname	Plocation	Hours
101	John	10	Alpha	Dallas	10
101	John	20	Beta	Houston	15
102	Mary	20	Beta	Houston	20
102	Mary	10	Alpha	Dallas	❌*Spurious*

Guideline 4

- Design relation schemas so that they can be joined using equality conditions on attributes that are appropriately related (primary key, foreign key pairs), in a way that guarantees no spurious tuples are generated.
- **Avoid relations** that contain matching attributes that are not (foreign key, primary key) pairs, because joining on such attributes may produce *spurious tuples*.

Summary Table

Problem	Cause	Example	Solution
Spurious Tuples	Joining on non-key attributes	Join on Plocation	Always join using Primary Key–Foreign Key pairs
Data Inconsistency	Extra or false records appear	Mary–Alpha example	Use proper schema decomposition
Correct Join	Lossless Join	EMPLOYEE–DEPARTMENT (join on Dnumber)	Ensure nonadditive (lossless) join property

Functional Dependencies

- To analyze and improve database design more **formally and scientifically**, we use the concept of **Functional Dependencies (FDs)**.
- FDs are the **foundation** for understanding **Normalization** — the process of designing good relational schemas.
- A **Functional Dependency (FD)** is a **constraint** between two sets of attributes in a relation.

Functional Dependencies

A functional dependency $X \rightarrow Y$ holds in a relation **R** if, for any two tuples t_1 and t_2 in **R**, whenever $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$.
That means — if we fix **X**, then **Y** cannot change.

Example Using Employee Table:
From the **semantics**, we can identify these **FDs**:
a. **Ssn → Ename**
Because a Social Security Number uniquely determines an employee's name.
b. **Pnumber → Pname, Plocation**
Because a project number uniquely determines its name and location.
c. **{Ssn, Pnumber} → Hours**
Because the combination of employee and project determines how many hours they work.

Ssn	Ename	Pnumber	Pname	Plocation	Hours
123	John	1	XProj	Delhi	10
124	Mary	2	YProj	Mumbai	12
123	John	2	YProj	Mumbai	8

Note:

- FD Is Not Always Reversible

If **X** → **Y**, it **does not mean Y** → **X**.

- **Example**:
  Ssn → **Ename** holds (Ssn determines employee name).
  But **Ename** → **Ssn** does *not* hold, because two employees could have the same name.

FDs Come from the Meaning of Data (Semantics):

- Functional dependencies reflect **real-world rules** or **business logic** — not just the current data in the table.
- For example:

- **{State, Driver_license_number} → Ssn**

means each driver license number (within a state) identifies a unique person.

- However, some dependencies can **change over time** —

For example:

- **Zip_code → Area_code** used to be true in the US, but not anymore (because phone area codes changed).

Note: Legal Relation States

A **legal relation state** (valid table) is one that **satisfies all the FDs** specified for the relation schema.

If a table violates even one FD, it is **not a legal state** for that schema.

FDs Can’t Be Discovered Automatically :

--You **cannot always find** FDs just by looking at the data.

Because FDs are **semantic constraints**, not always visible in current data.

However:

- You can **disprove** an FD with a single counterexample.

Example: TEACH(Teacher, Course, Text)

Let’s test if **Teacher** → **Course** holds.

No — because ‘Smith’ teaches two different courses.

Counterexample found → FD does **not** hold.

But **Teacher** → **Text** might hold (if each teacher uses one fixed textbook).

Teacher	Course	Text
Smith	Data Structures	Algo Made Easy
Smith	Data Management	DB Systems
Brown	Programming	Algo Made Easy

Real-world FD examples

1. College Database

Attribute	Meaning
Student_ID → Name, DOB, Department	A student’s ID uniquely identifies their personal details.
Course_ID → Course_Name, Credits	Each course ID corresponds to one course name and credit value.
Department → HOD	Each department has exactly one Head of Department.

2. Hospital Database

Attribute	Meaning
Doctor_ID → Doctor_Name, Specialty	A doctor’s ID uniquely determines their name and specialty.
Patient_ID → Patient_Name, DOB	Each patient ID identifies a unique person.
Room_Number → Ward_Type	Each room number corresponds to exactly one type of ward (e.g., ICU, General).

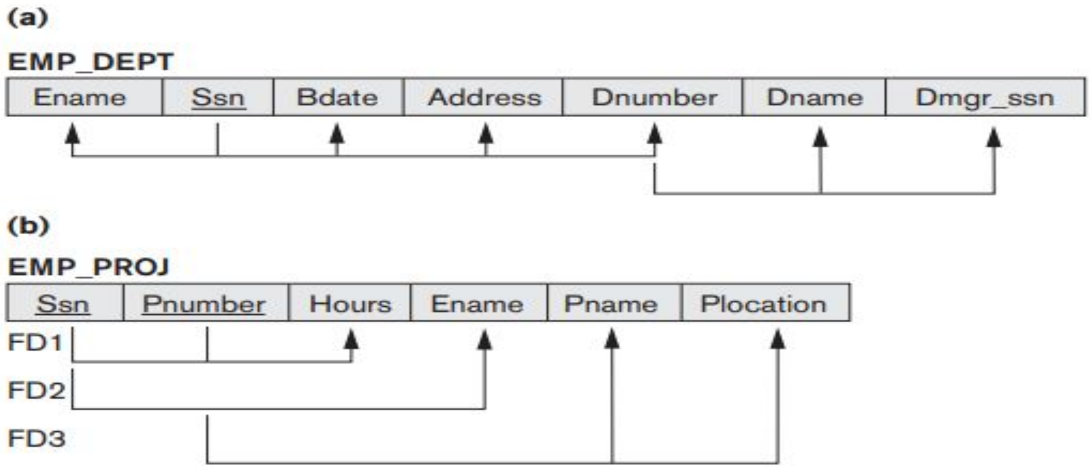

Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.

Consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. Ssn → Ename

b. Pnumber →{Pname, Plocation}

c. {Ssn, Pnumber} → Hours

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename)

(b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and

(c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

Why FDs Are Important

Functional dependencies help us:

- ❖ Detect redundancy in tables
- ❖ Decompose relations properly (Normalization)
- ❖ Preserve data consistency
- ❖ Understand the meaning and structure of data

Summary

Concept	Meaning
FD ($X \rightarrow Y$)	X determines Y; if two rows have same X, they must have same Y
Determinant	Left-hand side (X)
Dependent	Right-hand side (Y)
Legal Relation	A table that satisfies all FDs
FD from Key	If X is a key → $X \rightarrow R$
FDs from Semantics	Based on real-world meaning, not just current data
Violation	One counterexample disproves an FD
Importance	Foundation for Normalization (1NF, 2NF, 3NF, BCNF, etc.)

Example Relation: EMP_PROJ

Let's consider a relation (table):

EMP_PROJ(Ssn, Ename, Pnumber, Pname, Plocation, Hours)

Step 1: Identify the Functional Dependencies

From the **meaning (semantics)** of the attributes:

Ssn → Ename

- Each employee's Social Security Number identifies exactly one name.

(If Ssn = 101 → Ename = John)

Pnumber → Pname, Plocation

- Each project number identifies one project name and location.

(If Pnumber = 1 → Pname = Alpha, Plocation = Delhi)

{Ssn, Pnumber} → Hours

- The combination of employee and project determines how many hours the employee works on that project.

(If Ssn = 101 and Pnumber = 1 → Hours = 10)

Ssn	Ename	Pnumber	Pname	Plocation	Hours
101	John	1	Alpha	Delhi	10
101	John	2	Beta	Mumbai	8
102	Mary	1	Alpha	Delhi	12
103	Smith	3	Gamma	Pune	20

Ssn	Ename	Pnumber	Pname	Plocation	Hours
101	John	1	Alpha	Delhi	10
101	John	2	Beta	Mumbai	8
102	Mary	1	Alpha	Delhi	12
103	Smith	3	Gamma	Pune	20

(a) Ssn → Ename

- Two employees cannot share the same Ssn.
- Therefore, if two rows have the same Ssn, their Ename must also be the same.

Example:

Rows 1 and 2 both have Ssn = 101 → both have Ename = John

(b) Pnumber → {Pname, Plocation}

- Project number 1 always refers to project Alpha located in Delhi.
- Project number 2 always refers to project Beta located in Mumbai.
- So if Pnumber = 1, both Pname and Plocation are fixed.

{Ssn, Pnumber} together is the **primary key** for EMP_PROJ, because it uniquely identifies every row.

Concept	Meaning	Example
Functional Dependency	Relationship where one set of attributes uniquely determines another	Ssn → Ename
Determinant	Left-hand side of FD	Ssn, Pnumber
Dependent	Right-hand side of FD	Ename, Hours
Candidate Key	Set of attributes that uniquely identify a row	{Ssn, Pnumber}
Partial FD	When part of a composite key determines another attribute	Ssn → Ename
Full FD	When the whole composite key determines an attribute	{Ssn, Pnumber} → Hours

Full Functional Dependency (Full FD):

A **full FD** means: The *entire composite key* is necessary to determine another attribute — you cannot remove any part of the key and still determine that attribute.

Definition: If $X \rightarrow Y$, and no proper subset of X functionally determines Y, then $X \rightarrow Y$ is a **Full Functional Dependency**.

Example:
Consider relation: **ENROLL(Student_ID, Course_ID, Grade)**
Here, the **primary key** = (Student_ID, Course_ID)
FDs:
1.(Student_ID, Course_ID) → Grade (Full FD)
2.Student_ID → Name
3.Course_ID → Course_Name

•The **Grade** depends on **both** the student and the course together.
(You need both to know what grade a student got in a specific course.)
•You can't determine Grade by only Student_ID or only Course_ID.
→ Hence, it's a **Full FD**.

Partial Functional Dependency (Partial FD):

A **partial FD** means: Only a *part* of a composite key determines another attribute.

Definition: If $X \rightarrow Y$ and a proper subset of X also determines Y,
then $X \rightarrow Y$ is a **Partial Functional Dependency**.

Example of Partial FD

Again, relation: **ENROLL(Student_ID, Course_ID, Grade, Student_Name)**

Here:

- (Student_ID, Course_ID) → Grade (Full FD)
- (Student_ID, Course_ID) → Student_Name (Partial FD)

Because:

Student_ID → Student_Name (you don't need Course_ID to know the name).

So, Student_Name depends only on *part* of the key (Student_ID).
That's a **partial dependency**.

FD Real-time Example:
Think of FDs like “rules” of identity:
Each rule is a **Functional Dependency** — one thing determines another.

Determinant	**Determined By Rule**
Aadhaar Number	determines Person Name
Vehicle RegNo	determines Owner Name, Vehicle Type
RollNo	determines Student Name, Department
(RollNo, Subject)	determines Marks

Normal Forms Based on Primary Keys

- **Functional Dependencies (FDs)** can be used to check **how good or bad** a relation schema (table design) is — and **improve it** if needed. This improvement process is called **Normalization**.

What Is Normalization?

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

- Reduce **data redundancy** (avoid repeating the same data),
- Prevent **update anomalies** (inconsistencies during data changes),
- Ensure **data integrity** (data correctness).

- It is done by **analyzing FDs** and applying a series of **normal forms** (rules/conditions).
- Each **Normal Form (NF)** represents a level of improvement in design.

What Are Normal Forms?

---A **Normal Form** is a *set of conditions* that a relation must satisfy to be considered “well structured.”

- **1NF (First Normal Form):**
No repeating groups or multi-valued attributes.

- **2NF (Second Normal Form):**
No *partial dependency* on the primary key.

- **3NF (Third Normal Form):**
No *transitive dependency* on the primary key.

more advanced ones (BCNF, 4NF, 5NF, etc.).

How Normalization Works:

To normalize, we use:

1. **Functional Dependencies (FDs)** — which tell us how attributes depend on each other.
2. **Primary Key** — which uniquely identifies each tuple (row).

Then, we check:

- Whether attributes depend *fully* or *partially* on the key,
- Whether any attribute depends on a *non-key* attribute,
- And we decompose (split) the table if necessary.

How Relations Are Designed :

There are **two main ways** database designers create relations:

- ❖ **Conceptual Schema Design (using ER or EER models)**
  - Top–Down Approach
  - Designers start with a **conceptual model** like an **ER diagram**, then map entities and relationships to relational tables.

Example:

- ER Diagram → Employee, Department, Project → converted into tables.

- ❖ **Existing Data Design (from real-world files/forms)**
  - Bottom–Up Approach
  - Designers may start from **existing files, spreadsheets, or reports**, and convert them into relations.

Example:

- Old payroll system or Excel sheets → converted into relational tables.

Why We Need Normalization?

Even after designing relations (from ER model or existing system), we must **evaluate** their quality:

- Is there **redundancy**?
- Can the table cause **update, insertion, or deletion anomalies**?
- Are dependencies properly represented?

To ensure good design, we **apply the rules of normal forms** —
we test the relation against 1NF, 2NF, and 3NF, and **decompose** it if necessary.

The Idea Behind Normal Forms:
Each Normal Form removes certain types of problems.

Normal Form	Type of Problem Removed	Dependency Type
1NF	Repeating groups / multivalued attributes	None (basic structure)
2NF	Partial dependency (attribute depends only on part of a key)	Partial FD
3NF	Transitive dependency (attribute depends on non-key attribute)	Transitive FD

Note:
Attribute: A column in a table.
Tuple: A row in a table.
Primary Key: A minimal set of attributes that uniquely identifies each row.
Candidate Key: Any attribute (or set of attributes) that can uniquely identify a row.
Non-prime attribute: An attribute that is *not part of any candidate key*.
Functional Dependency (FD): A $\rightarrow$ B means the value of A determines the value of B.

First Normal Form (1NF):

A relation is in **First Normal Form (1NF)** if:
Every attribute contains only *atomic (indivisible)* values —
i.e., no repeating groups, arrays, or sets.

Not in 1NF:
Because **Subjects** contains multiple values (not atomic).

StudentID	Name	Subjects
101	John	{Math, Physics}

1NF:
Now, all values are atomic (one value per cell).

StudentID	Name	Subject
101	John	Math
101	John	Physics

Second Normal Form (2NF):

A relation is in **Second Normal Form (2NF)** if:

1. It is already in **1NF**, and
2. There is **no partial dependency** of a *non-key attribute* on *part of a composite key.*

What Is Partial Dependency?

A **partial dependency** occurs when:

A non-key attribute depends on only *part* of a composite primary key.

Example

Relation: **EMP_PROJ(Ssn, Pnumber, Ename, Pname, Hours)**

Primary Key: **(Ssn, Pnumber)**

FDs:

1. Ssn → Ename
2. Pnumber → Pname
3. {Ssn, Pnumber} → Hours

- Ename depends only on **Ssn** (part of key) → partial dependency
- Pname depends only on **Pnumber** (part of key) → partial dependency
- Hours depends on **both (Ssn, Pnumber)** → full dependency

Hence, **EMP_PROJ** is **not in 2NF**.

Convert to 2NF:

Split into two relations:

1. **EMP(Ssn, Ename)**
2. **PROJ(Pnumber, Pname)**
3. **WORKS_ON(Ssn, Pnumber, Hours)**

Now:

- No partial dependencies exist.
- Each table has attributes that depend *fully* on its key.

Third Normal Form (3NF):

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

1. It is already in **2NF**, and
2. It has **no transitive dependency** of non-key attributes on the primary key.

What Is Transitive Dependency?

A **transitive dependency** occurs when:

A non-key attribute depends on *another non-key attribute*, which depends on the key.

That is:

$\mathbf{A \rightarrow B \rightarrow C} \Rightarrow A \rightarrow C$ (transitively)

Example

Relation: **STUDENT(RollNo, DeptNo, DeptName, HOD)**

FDs:

1. RollNo → DeptNo, DeptName, HOD
2. DeptNo → DeptName, HOD

Here:

- RollNo → DeptNo → DeptName, HOD

⇒ RollNo → DeptName, HOD (transitive dependency)

Convert to 3NF:

Split into:

1. **STUDENT(RollNo, DeptNo)**
2. **DEPARTMENT(DeptNo, DeptName, HOD)**

Now there is no transitive dependency — 3NF achieved

Normal Form	Condition	Problem Removed	Example Fix
1NF	All attributes atomic	Repeating groups	Split multivalued columns
2NF	1NF + no partial dependency	Partial dependency	Split based on full key dependency
3NF	2NF + no transitive dependency	Transitive dependency	Separate dependent non-key attributes

1NF: Make sure data is stored in simple, atomic form.
Each cell = one value.

- **2NF:** Make sure every non-key attribute depends on the *whole* key.
Avoid partial dependencies.
- **3NF:** Make sure non-key attributes depend *only* on the key, not other non-keys.
Avoid transitive dependencies.

Boyce-Codd Normal Form

Boyce–Codd Normal Form (BCNF) is an advanced version of **Third Normal Form (3NF)**.
It removes **all remaining anomalies** that 3NF may still allow.

Every relation in **BCNF** is also in **3NF**,
but not every **3NF** relation is in **BCNF**.

Why BCNF is needed (problem with 3NF)
Even after achieving **3NF**, some redundancy can still remain.

Formal Definition of BCNF
A relation schema **R** is in **BCNF** if for **every non-trivial functional dependency**
$X \rightarrow A$ that holds in R,
X must be a superkey of R.

Meaning:

- **Non-trivial FD**: means A is not part of X.
  (Example: Area → County_name is non-trivial.)
- **Superkey**: A set of attributes that can uniquely identify all tuples in a relation.
  So, **in BCNF**, every determinant (the left side of an FD) must be a superkey.

BCNF — Boyce-Codd Normal Form

Rule: Stronger version of 3NF

- For every functional dependency **X → Y**,

X must be a candidate key.

Meaning: If a non-key attribute determines something else, it violates BCNF.

- **Example:**

FDs: (Teacher, Subject) → Room

Room → Teacher

Here, Room → Teacher means Room determines Teacher (but Room is not a key)

So not in BCNF.

Decompose: into tables based on these dependencies.

Teacher	Subject	Room
A	DBMS	R1
B	Java	R1

Normalization of Relations – Summary

Definition:
Normalization is the process of organizing data in a database to reduce redundancy and avoid anomalies (update, insertion, deletion).

Objectives:

- Eliminate data redundancy
- Ensure data integrity
- Simplify maintenance
- Achieve efficient storage & access

Normal Form	Removes	Key Rule / Condition
1NF	Repeating groups	Each field contains atomic (single) values
2NF	Partial dependency	Non-key attributes depend on the whole key
3NF	Transitive dependency	Non-key attributes depend only on key
BCNF	Remaining anomalies	Every determinant is a superkey
4NF/5NF	MVDs & Join dependency	No multivalued or join dependency

Result: Well-structured, consistent, and redundancy-free database design.

General Definitions of Second and Third Normal Forms

- ## Second Normal Form (2NF):

A relation is in 2NF if it is in 1NF and every non-prime attribute is fully functionally dependent on the entire primary key (no partial dependency).
→ Removes partial dependency anomalies.
→ Ensures attributes depend on the whole key, not just part of it.

- ## Third Normal Form (3NF):

A relation is in 3NF if it is in 2NF and there is no transitive dependency — that is, no non-prime attribute depends on another non-prime attribute.
→ Removes transitive dependency anomalies.
→ Ensures non-key attributes depend only on candidate keys.

Result: 2NF and 3NF help create a well-structured database free from partial and transitive dependencies.

Normal Form	Condition	Problem Removed
1NF	Atomic values only	Repeating groups
2NF	No partial dependency	Redundancy due to part-key dependency
3NF	No transitive dependency	Redundancy due to non-key dependency
BCNF	Every determinant is a candidate key	Anomalies in complex dependencies
4NF	No multi-valued dependency	Redundant multi-valued facts
5NF	No join dependency	Data loss during joins

Properties of Relational Decompositions

- When we normalize a relation (table), we often **decompose** it into smaller relations to remove redundancy and anomalies.
- However, **just decomposing** into 3NF or BCNF is **not enough** — the resulting set of relations must also satisfy certain **important properties** to ensure a *good database design*.

Decomposition means breaking one large relation (table) into two or more smaller relations.

- Goal: Remove **redundancy** and **update anomalies**, and make the design more efficient.

Example:

- EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation)
- can be decomposed into:
  - EMP(Ssn, Ename)
  - PROJECT(Pnumber, Pname, Plocation)
  - WORKS_ON(Ssn, Pnumber, Hours)

Why Normal Forms Are Not Enough

Even if each decomposed relation is in **3NF or BCNF**, it might still be a **bad design** if:

- It **loses information** when joined back (spurious tuples appear), or
- It **loses dependencies**, meaning some functional dependencies are no longer enforceable.

Hence, we need **extra properties** beyond just normal forms.

Dependency Preservation Property

- Definition:

A decomposition is **dependency-preserving** if every functional dependency in F can be found (directly or indirectly) in one of the decomposed relations.

- If not preserved, we must join tables to check some constraints — which is costly and inefficient.

Key Properties of a Good Decomposition

Property	Meaning	Purpose
Attribute Preservation	All attributes from the original relation must appear in at least one of the decomposed relations.	Prevent loss of data (no attribute should disappear).
Dependency Preservation	All functional dependencies from the original relation must be represented in the decomposed relations.	Allows constraints to be enforced without needing joins.
Nonadditive (Lossless) Join Property	Joining the decomposed relations should recreate the original relation exactly — without adding spurious (fake) tuples.	

Summary

A **good decomposition** must ensure:

1. **Attribute Preservation** – No data loss
2. **Dependency Preservation** – Constraints can be checked easily
3. **Nonadditive Join (Lossless)** – No spurious tuples

These properties ensure that normalization results in a **sound and practical** relational database design.