Normalisation Part - 1

What is Database Schema?

Database Schema is a blueprint of the actual database that you are going to make. This is not the actual database, it is the design or the blueprint of how actually the data will look like.

Analogy: Think of a database schema like the architectural blueprint of a house. The blueprint shows where rooms will be, how they connect, what size they are, but it's not the actual house - it's just the plan.

What Database Schema contains:

- → How we will store data
- → Structure of tables
- → How tables are associated with each other
- → Relations between tables
- → Rules and constraints
- → Data types
- → Keys

Design Process:

When we are actually designing the database schema, we don't just tell about the tables, but we also tell about how the tables are associated with each other, kind of like the relations between the tables.

This database schema, or the blueprint, is first of all, definitely coded on a pen and paper so that you actually understand what are the nuances that you actually need to cover.

→ Traditional way

Example of Database Schema Design:

School Database Schema:

- Students Table (student_id, name, class_id, age)
- Classes Table (class_id, class_name, teacher_id)
- Teachers Table (teacher_id, teacher_name, subject)
- Relationships:
  - Students.class_id → Classes.class_id
  - Classes.teacher_id → Teachers.teacher_id

Modern Schema Creation:

But now in modern backend frameworks, we can define a schema in our backend code and then the framework automatically helps us to create a database out of it so you don't have to manually do all of this. So there are features which actually help you to just define the schema programmatically. And then that schema gets replicated as actual database.

In modern backend frameworks, you can define schemas programmatically:

```
// Example in a framework
const studentSchema = {
  student_id: { type: 'INTEGER', primaryKey: true },
  name: { type: 'VARCHAR(50)', notNull: true },
  class_id: { type: 'INTEGER', foreignKey: 'classes.class_id' }
};
```

The framework then automatically creates the actual database tables from this schema.

What is Database Instance?

Database Instance is like a snapshot or photograph of your actual database at any specific moment in time. It's the real, living database with actual data stored in it.

Definition: Database instance is just a snapshot of the actual database that exists. It is the actual database that you have prepared in the DBMS at any point of time.

Database Schema	Database Instance
Blueprint / Design	Actual Database
Shows structure	Contains real data
Doesn't change often	Changes frequently
Like a house plan	Like the actual house with people living in it.

Example:

- Schema: "We will have a students table with columns: id, name, age"

```
-- Schema defines this structure
CREATE TABLE Students (
    Student_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT
);
```

- Instance: The actual Students table with real data like "1, John, 16"

-- Instance is the actual data at any moment:

Student_id	name	age	
1	John	16	this is part of the current database instance
2	Sarah	17	
3	Mike	15	

Functional Dependency

Important Note: In RDBMS, we refer to columns attributes and rows as tuples.

What is functional dependency?

Functional dependency defines the relationship between two attributes (columns). It's like saying "If you know this, you can uniquely determine that."

Notation: $X \rightarrow Y$

This means "Y is dependend on X" or "Y depends on X"

- Y is the dependent attribute
- X is the determinant attribute

Simple Definition: Y depends on X means that for every valid value of X, we can uniquely identify Y.

Analogy: Your student ID determines your name. If I know your student ID, I can find exactly one name that belongs to that ID.

Example with Employee Table:

emp_id	emp_name	emp_salary
101	Rahul	55000
102	Priya	62000
103	Arjun	48000

Valid Functional Dependencies:

- emp_id → emp_name ✓ (Employee name depends on Employee ID)
- emp_id → emp_salary ✓ (Employee salary depends on Employee ID)

Invalid Functional Dependency:

- emp-name → emp-salary ✗ (Two employees can have the same name, so you can't uniquely determine salary from name alone)

Why is emp-name → emp-salary invalid?
Because multiple employees might have the same name:

emp-id	emp-name	emp-salary
101	John	50000
102	John	60000

If you only know the name is "John", you can't determine if the salary is 50000 or 60000.

Varifying Functional Dependencies

Question: Can we programmatically check if a table follows functional dependencies in MySQL?

Answer: We want to have a query or assertion. SQL standards (SQL-92) suggest using ASSERTIONS:

```
CREATE ASSERTION emp-name-dependency
CHECK (NOT EXISTS (
        SELECT *
        FROM Employee AS E1, Employee AS E2
        WHERE E1.emp_id = E2.emp_id AND
              E1.emp_name <> E2.emp-name
));
```

What this query checks:

- There should be no two rows where employee ID is the same but employee name is different.
- This would validate the functional dependency emp-id → emp-name.

Reality check:

- This assertion feature is mentioned in SQL-92 standard.
- Most databases (MySQL, postgreSQL, Oracle) do NOT support assertions
- Only IBM DB2 supports some assertions features for validation checks.
- Conclusion: In MySQL, there's no direct way to programmatically check functional dependencies.

Why study functional dependencies then?

Even though we can't verify them programmatically, functional dependencies help us:

1. Design better database schemas
2. Identify table keys and candidate keys
3. Detect database anomalies (update, insert, delete anomalies)
4. Perform normalization (creating efficient database designs)
5. Avoid data redundancy and inconsistencies
6. Make overall schema better and efficient.

Axioms (Armstrong's Axioms)

Axioms are basic rules or principles that are accepted as true without proof. In database theory, Armstrong's axioms help us work with functional dependencies.

Analogy: Axioms are like basic math rules (like 2+2=4) that everyone agrees are true and uses to solve bigger problem.

The three Basic Armstrong's Axioms :

1. Reflexivity Axiom (Trivial Dependency)
   - Rule: If Y is a subset of X, then $X \rightarrow Y$.
   - Simple Meaning: A set of attributes always determines its subsets
   - Example: If we have {emp_id, emp_name} then {emp_id, emp_name} $\Rightarrow$ emp_id.
   - Real-life Example: Your full address can determine your city (city is part of address)

2. Augmentation Axiom
   - Rule: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ (for any attribute Z)
   - Simple Meaning: If X determines Y, then adding the same attribute to both sides keeps the dependency valid.
   - Example: If emp_id $\rightarrow$ emp_name, then {emp_id, dept_id} $\rightarrow$ {emp_name, dept_id}
   - Real-life Example: If student ID determines name, then student ID + course determines name + course.

3. Transitivity Axiom
   - Rule: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
   - Simple Meaning: If A leads to B, and B leads to C, then A leads to C
   - Example: If emp_id $\rightarrow$ dept_id and dept_id $\rightarrow$ dept_name, then emp_id $\rightarrow$ dept_name.
   - Real-life Example: If Student_ID $\rightarrow$ Class_ID and Class_ID $\rightarrow$ Teacher_name then Student_ID $\rightarrow$ Teacher_name.

Derived Rules (From Basic Axioms):

4. Union:
   - Rule : If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
   - Example: If emp_id → name and emp-id → salary, then emp-id → {name, salary}

5. Decomposition
   - Rule: If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$
   - Example: If emp-id → {name, salary} then emp_id → name and emp-id → salary.

6. Pseudo-Transitivity Rule
   - Rule: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$
   - Example: Complex rule used in advanced database design.

What is Transitive Dependency?

A transitive dependency occurs when a non-prime attribute depends indirectly on a candidate key through another non-prime attribute.

In simple terms:

If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$ is a transitive dependency.

But this is only a problem when C is a non-prime attributes and B is also non-prime.

Example: Bad Design

Let's say we have this table:

emp id	dept id	dept_name	emp-salary
101	D1	Sales	50000
102	D1	Sales	55000
103	D2	IT	60000

Candidate Key: emp-id

Dependencies:

- emp_id → dept_id (direct dependency)
- dept_id → dept_name (direct dependency)
- So: emp_id → dept_name (transitive dependency)

Problem:

- Redundancy: "Sales" repeats for every employee in D1
- Update anomaly: If "Sales" becomes "Global Sales", you must update multiple rows.
- Insert anomaly: You can't add a new department unless an employee exists.
- Delete anomaly: Deleting all employees in D1 removes "Sales" info.

Good design: Split the table into two:

Employee Table:

emp-id	dept_id	emp_salary
101	D1	50000
102	D1	55000
103	D2	60000

Department table

dept_id	dept_name
D1	Sales
D2	IT

Now:

- dept_name depends directly on dept_id
- No transitive dependency
- Clean, efficient, and easy to maintain

Prime and Non-Prime Attributes

→ Prime Attributes: Attributes that are part of any candidate key (not just primary key, but any candidate key in the table)

→ Non-Prime Attributes: Attributes that are NOT part of any candidate key in the table.

Note: We consider all candidate keys, not just the primary key

Example:

consider a table with attributes {A, B, C, D} and candidate keys {A, B} and {A, C}

Prime Attributes: A, B, C (because they appear in candidate keys)

Non-Prime Attributes: D (doesn't appear in any candidate key)

Fully and Partial Dependency

→ Full Dependency: When a non-prime attributes depends on the complete primary key (all parts of a composite key)

Y is fully dependent on X if:

1. Y depends on X ($X \rightarrow Y$ holds)
2. Y cannot depend on any proper subset of X
3. All parts of X are necessary to determine Y

Y needs all parts of X to be uniquely determined. If you remove any part of X, you cannot determine Y anymore.

→ Partial Dependency : When a non-prime attribute depends on only part of the primary key (subset of composite key).

Y is partially dependent on X if :

1. X is a composite key (has multiple attributes)
2. Y depends on some proper subset of X (not the full X)
3. Y can be determined by only part of X

Y only needs part of X to be uniquely determined, not the complete X.

Example:

Enrollment table with composite key {student_id, course_id}

Student_id	course_id	student name	course-name	instructor	grade
1	C1	John	Math	Dr. Smith	A
1	C2	John	Science	Dr. Jones	B
2	C1	Mary	Math	Dr. Smith	B
2	C2	Mary	Science	Dr. Jones	A

Full Dependencies (need both student_id AND course_id) :

- {student_id, course_id} → grade ✓
  - → Cannot determine grade with just student_id (John has different grades in different courses)
  - → Cannot determine grade with just course_id (different students have different grades in same course)
  - → Need BOTH to determine the grade.

Partial Dependencies (need only part of the composite key) :

- student_id → student_name ✗ (Partial dependency)
  - → Only need student_id, don't need course_id
  - → John's name is same regardless of course

- course_id → course_name ✗ (Partial Dependency)
  - → Only need course_id, don't need student_id
  - → Mam course name is same regardless of student

- course_id → instructor ✗ (Partial Dependency)
  - → Only need course_id to know who teaches that course

Why Partial Dependencies are Problem:

1. Data Redundancy:
   - Student name repeats for every course enrollment
   - Course name repeats for every student enrollment
2. Update Anomaly: If John changes his name, need to update multiple rows.
3. Insert Anomaly: Cannot store course information without a student enrollment.
4. Delete Anomaly: If all students drop a course, we lose course information.

Solution to Partial Dependencies:

Break into separate tables:

- Students: student_id, student_name
- Course: course_id, course_name, instructor
- Enrollments: student_id, course_id, grade

Database Keys

A Key in a database is an attribute (or a set of attributes) that helps uniquely identify a record (row) in a table. Key also help enforce rules and connect tables together.

Types of Keys:

1. Super Key:

Defination: Any set of attributes that can uniquely identify a row in a table (may include extra unnecessary attributes)

Key Properties:

→ contains at least one candidate key

→ May have redundant attributes that are not needed for uniqueness

→ If we remove some attributes and it still uniquely identifies rows, those removed attributes were redundant.

Example:

In a student table with attributes {Student_id, name, email, phone}:

- {Student_id} - Super Key
- {Student_id, name} - Super Key
- {Email} - Super Key
- {Student_id, name, email} - Super Key

2. Candidate Key:

Defination: A minimal super key - smallest set of attributes that can uniquely identify a record.

Key Properties:

→ Must be unique (no duplicates)

→ Must be minimal (cannot remove any attributes and still maintain uniqueness)

→ cannot contain NULL values

→ A table can have multiple candidate keys.

Example:

From the super keys above:

- {student_id} - Candidate Key (minimal)
- {email} - Candidate Key (minimal)
- {student_id, name} - NOT candidate Key (not minimal, can remove name)

3. Primary Key

Defination: One candidate key chosen as the main identifier for the table.

Rules for Primary Key:

→ Cannot be NULL

→ Must be unique

→ Should not change frequently

→ Preferably simple (single attribute)

Example:

Choose {student_id} as primary key from candidate keys {student_id, email}

4. Alternate Key

Definition: Candidate Keys that are NOT chosen as the primary key

Key Properties:

→ Were eligible to be primary key but not selected

→ Still maintain uniqueness constraint

→ Can be used to create unique indexes

→ Provide alternate ways to access record

Example: If {student_id} is primary key, then {email} becomes alternate key.

5. composite Key

Defination: A candidate Key made up of two or more columns that together uniquely identify a row in a table.

Key properties:

→ No single column alone can guarantee uniqeness

→ But the combination of column does

→ Also called compound Key

Example:

In an Enrollment table:

- Primary Key: { student_id, course_id }
- Neither student_id alone nor course_id alone can uniquely identify an enrollment
- But together they do: One student can't enroll in the same course twice.

6. Foreign Key:

Definition: A key that refers to the primary key of another table.

Purpose: Creates relationships between tables and maintains referential integrity.

Key Properties:

→ Links two tables together

→ Value must exist in the referenced table's primary key (or be NULL if allowed)

→ Prevents orphaned records (records that reference non-existent data)

→ can have duplicate values (unlike primary key)

Example:

Students Table:

student_id	name	class_id → Foreign Key
1	John	101
2	Sarah	102

(student_id: Primary Key)

Classes Table:

class_id	class name
101	Math
102	Science

(class_id: Primary Key)

7. Unique Key

Definition: Ensures all values in a column (or combination of columns) are different from each other.

Key properties:

→ Allows NULL values (unlike Primary Key) but only one NULL per column

→ Prevents duplicate non-NULL values

→ A table can have multiple unique keys

→ Automatically creates an index for faster searches.

Difference from Primary Key:

Primary Key	Unique Key
• Only one per table	• Multiple allowed per table
• cannot be NULL	• Can be NULL (one NULL allowed)
• Clustered index	• Non-clustered index

Example:

student_id	email	phone	ssn	name
1	john@mail.com	9876543210	123-45-6789	John
2	mary@mail.com	9876543211	987-65-4321	Mary
3	NULL	9876543212	456-78-9123	Sam

```
CREATE TABLE Students {
    Student_id INT PRIMARY KEY,     -- Primary Key (cannot be NULL)
    email VARCHAR(100) UNIQUE,      -- Unique Key (can be NULL)
    phone VARCHAR(15) UNIQUE,       -- Another Unique Key
    ssn VARCHAR(11) UNIQUE,         -- Another Unique Key
    name VARCHAR(50)                -- Regular column
);
```

Note: Sam can have NULL email (unique constraint allows one NULL), but all phone numbers and SSNs must be different.

Functional Dependency closure

Functional Dependency closure (F+) is the complete set of all functional dependencies that can be derived from a given set of functional dependencies using Armstrong's axioms.

Steps to find closure:

1. Start with given functional dependencies
2. Apply reflexivity (every attribute determines itself)
3. Apply transitivity to find indirect dependencies
4. Apply union and decomposition as needed
5. Continue until no new dependencies can be derived

Example:

Given: $F = \{A \rightarrow B, B \rightarrow C\}$

Step-by-step derivation of F+:

1. Start with Given Dependencies:
   - $A \rightarrow B$
   - $B \rightarrow C$

2. Apply Transitivity :
   - since $A \to B$ and $B \to C$, then $A \to C$

3. Apply Reflexitivity :
   - $A \to A$
   - $B \to B$
   - $C \to C$

4. Apply Augmentation :
   - From $A \to B$, we get $AB \to BB$, which simplifies to $AB \to B$
   - From $A \to B$, we get $AC \to BC$
   - And many more...

5. Complete closure F+ :

F+ = {

$A \to A$, $B \to B$, $C \to C$, (Reflexivity)

$A \to B$, $B \to C$, $A \to C$, (Given + Transitivity)

$A \to AB$, $A \to AC$, $A \to BC$, (Union)

$AB \to A$, $AB \to B$, $AB \to C$, (Decomposition)

$AC \to A$, $AC \to B$, $AC \to C$,

$BC \to B$, $BC \to C$

$ABC \to A$, $ABC \to B$, $ABC \to C$

... and more

}

Practical Use:

Finding closure helps us :

1. Verify if a dependency exists
2. Find all possible keys
3. Check for redundant dependencies

Attribute closure

Attribute closure (X+) is the set of all attributes that can be functionally determined by a given set of attributes X.

Algorithm to find Attribute closure:

1. Start with the given attribute
2. Look for functional dependencies where the left side is what we current have
3. Add the right side attribute to our set
4. Repeat until no new attributes can be added

Example:

Given:

- Attributes: {A, B, C, D, E}
- Dependencies: {A → B, A → E, C → B, C → E, B → D}

Find A+ (closure of A):

Step 1: Start with A
current set: {A}

Step 2: Find dependencies with A on left side
- A → B (add B)
- A → E (add E) current set: {A, B, E}

Step 3: Find dependies with B or E on left side
- B → D (add D) current set: {A, B, D, E}

Step 4: Check for more dependencies
No more dependencies can be applied.

Result: $A^+ = \{A, B, D, E\}$

Another Example : Find C^+ :

Step 1 : Start with C
current set : {C}

Step 2 : Apply $C \to B$ and $C \to E$
current set : {C, B, E}

Step 3 : Apply $B \to D$
current set : {C, B, D, E}

Result : $C^+ = \{C, B, D, E\}$

Uses of Attribute closure :
1. Check if $X \to Y$: $Y \subseteq X^+$?
2. Find candidate keys : If X^+ = all attributes, then X is a super key
3. Verify functional dependencies

Numericals

→ Problem 1 :
Given : R = {A, B, C, D, E}
and FD : $A \to B$, $A \to E$, $C \to B$, $E \to C$, $B \to D$

Solution :
Step 1 : RHS and forced attributes
RHS set = {B, E, D}
Attributes not on RHS = {A, C}
Therefore A and C must be included in every candidate key.

Step 2 : compute closure of the forced set {A, C}
Start : $(AC)^+ = \{A, C\}$

Use FDs:

- $A \to B, E \Rightarrow$ add B, E. Now $\{A, B, C, E\}$
- $B \to D \Rightarrow$ add D. Now $\{A, B, C, D, E\} = R$

So, $(AC)^+ = R$. Thus $\{A, C\}$ is a superkey

Step 3: Check minimality

check singletons:

- $A^+ = \{A, B, E, D\}$ (misses C) → not key
- $C^+ = \{C, B, E, D\}$ (misses A) → not key.

Since neither A nor C alone is a key, $\{A, C\}$ is minimal

Answer: Candidate Key = $\{A, C\}$

→ Problem 2:

Given: $R = \{A, B, C, D\}$

and FDs: $AB \to C$, $BC \to D$, $CD \to A$

Step 1: RHS and forced attributes

RHS set = $\{C, D, A\}$

Attributes not on RHS = $\{B\}$

Therefore B must appear in every candidate key.

Step 2: Try combination containing B (compute closure)

1. $(AB)^+$

Start $\{A, B\}$

- $AB \to C \Rightarrow$ add $C \to \{A, B, C\}$
- $BC \to D$ (we have B and C) $\Rightarrow$ add $D \to \{A, B, C, D\} = R$

So, $(AB)^+ = R \Rightarrow AB$ is a superkey. check if minimal: neither A nor B alone gives whole R, so AB is a candidate key.

2. $(BC)^+$

Start $\{B, C\}$

- $BC \rightarrow D \Rightarrow$ add $D \rightarrow \{B, C, D\}$
- $CD \rightarrow A$ (we have C and D) $\Rightarrow$ add $A \rightarrow \{A, B, C, D\} = R$

So, $(BC)^+ = R$. check minimality : neither B nor C alone is a key, so BC is a candidate key.

3. Other combination with B :

- BD closure does not generate A or C without additional FDs firing, so BD is not a key.
- Any superset of AB or BC is not minimal

Answer: Candidate Keys = $\{AB, BC\}$

Normalisation Part - 2

What is Redundancy?

Redundancy is like having duplicate or unnecessary copies of the same information stored in multiple places in your database. It's like writing the same place number in multiple address books.

Example: Bad Example - Student course Table:

Student_id	Student_name	course-id	course_name	instructor
1	John	C1	Math	Dr. Smith
1	John	C2	Science	Dr. Jones
2	Mary	C1	Math	Dr. Smith
2	Mary	C3	History	Dr. Brown

Redundant data:

→ "John" is repeated twice (redundant student name)

→ "Mary" is repeated twice (redundant student name)

→ "Math" and "Dr. Smith" are repeated twice (redundant course info)

Problem caused by Redundancy

1. Storage Waste:
   - Takes up necessary space
   - Larger databases cost more to maintain.

2. Update Anomaly:
   - If John changes his name, you need to update multiple rows
   - If you miss updating one row, you have inconsistent data

3. Insert Anomaly:
   - Cannot add a new course without enrolling a student
   - cannot add student information without enrolling in a course.

4. Delete Anomaly:

- If John drops all courses, you lose his information completely
- If all students drop math, you lose the course information.

5. Data Inconsistency:

- Same information might be stored differently in different places
- Example: "Dr. Smith" vs "Dr. John Smith" for the same instructor

What is Normalization?

Normalization is the process of determining how much redundancy exists in a table and gives us techniques to reduce it.

Goals of Normalization:

1. Eliminate data redundancy (remove duplicate data)
2. Reduce storage space (more efficient database)
3. Prevent data inconsistencies (avoid conflicting information)
4. Make updates easier (change data in one place only)
5. Improve data integrity (maintain accurate, reliable data)

The Process: Normalization involves splitting large tables into smaller, related tables and defining relationships between them to eliminate redundancy while preserving data integrity.

Normal Forms

Normal forms actually help you understand what level of redundancy you have. And they give you techniques to actually reduce the redundancy at specific positions.

Hierarchy of Normal Forms :

1NF (First Normal Form)

↓

2NF (Second Normal Form)

↓

3NF (Third Normal Form)

↓

BCNF (Boyce-Codd Normal Form)

↓

4NF (Fourth Normal Form)

↓

5NF (Fifth Normal form)

Important Rules :

- Every normal form is dependent on the previous normal form
- A table in 3NF automatically in 2NF and 1NF
- You cannot skip levels — must acheive 1NF before 2NF, 2NF before 3NF etc.

Common Normal Forms used :

- 1NF, 2NF, 3NF - Used in most real-world applications
- BCNF - used for more complex scenarios
- 4NF, 5NF - Used in advanced database design (rarely needed)

First Normal Form (1NF)

Any attribute must only contain atomic values. Atomic values means indivisible - cannot be broken down into smaller meaningful parts. Each cell in a table should contain only ONE piece of information.

Why 1NF is important?

- Enables proper querying - can search for specific values
- Supports indexing - Database can create efficient indexes
- Allows sorting - can order data properly
- Prevents confusion - Each cell has clear, single meaning

Rules for 1NF:

1. Each column should contain atomic (indivisible) values
2. No repeating groups or arrays in a single column
3. Each row should be unique
4. Each column should have a unique name.

Example of 1NF Violation (Bad design):

Students Table (NOT in 1NF):

Student-id	student_name	courses	phone-numbers
1	John	Math, Science History	123-456-7890, 987-654-3210
2	Mary	Math, English	555-123-4567
3	Sam	Science	111-222-3333, 444-555-666, 777-888-9999

Problem with this Design:

1. Non-Atomic Values:
   - course column contains multiples values separated by commas.
   - phone-numbers contains multiple phone numbers
   - cannot easily search for students taking "Math" only.

2. Query Difficulties:

-- How do you find all students taking Math? This won't work properly:

```
SELECT * FROM students WHERE courses = 'Math'; -- won't find John
```

-- You'd need complex string operations:

```
SELECT * FROM students WHERE courses LIKE '%Math'; -- Finds "Mathematics" too!
```

3. Storage Inconsistency:

- Some students have 1 phone numbers, others have 2 or 3
- Database cannot enfore consistent structure.

4. Update Problems:

- To remove "History" from John's courses, need string manipulation
- To add a phone number, need to modify existing string.

5. Sorting Issues:

- Cannot sort by individual course names
- Cannot sort by phone numbers properly.

How to Convert to 1NF (Good Design)

Solution 1: Separate Rows for each value

student_Phone_table:

Student_id	student_name	phone_numbers
1	John	123-456-7890
1	John	987-654-3210
2	Mary	555-123-4567
3	Sam	111-222-333
3	Sam	444-555-666
3	Sam	777-888-999

Student_Subject_Table :

Student_id	Student_name	subject
1	John	Math
1	John	Science
1	John	History
2	Mary	Math
2	Mary	English
3	Sam	Science

Solution 2: Separate Tables (Better Approach) → Recommended

Students Table :

Student_id	Student_name
1	John
2	Mary
3	Sam

Student_courses Table :

Student_id	course-name
1	Math
1	Science
1	History
2	Math
2	English
3	Science

Student_Phones Table :

Student_id	phone-number
1	123-456-7890
1	987-654-3210
2	555-123-4567
3	111-222-3333
3	444-555-6666
3	777-888-9999

Benefits of INF Design :

1. Easy Queries :

```
-- Find all students taking Math:
SELECT s.student_name
FROM students s
JOIN student_courses sc ON s.student_id = sc.student_id
WHERE sc.course_name = 'Math';

-- Find all phone numbers for John
SELECT sp.phone_number
FROM students s
JOIN students_phones sp ON s.student_id = sp.student_id
WHERE s.student_name = 'John';
```

2. Easy Updates :

```
-- Remove History from John's courses:
DELETE FROM student_courses
WHERE student_id = 1 AND course_name = 'History';

-- Add new phone number for Mary :
INSERT INTO student_phones VALUES (2, '999-888-7777');
```

3. Data Integrity :

- Each course is stored consistently
- Each phone number is stored in standard format
- Can add constraints to validate data

4. Flexibility:

- Can easily add new courses or phone numbers
- can add additional attributes to courses (like credits, instructor)
- Can enforce business rules (like maximum 3 phone numbers per student)

Second Normal Form (2NF)

A table is in 2NF if:

1. It is already in 1NF
2. It has no partial dependencies

Partial Dependency: When a non-prime attribute depends on only part of a composite primary key (not the entire key).
Every non-key column should depend on the entire primary key, not just part of it.

Example of 2NF violation (Bad design):

Course_Enrollment Table (NOT in 2NF):

student-id	course-id	student_name	student-age	course-name	instructor	grade
1	C1	John	20	Math	Dr. Smith	A
1	C2	John	20	Science	Dr. Jones	B
2	C1	Mary	21	Math	Dr. Smith	B
2	C3	Mary	21	History	Dr. Brown	A
3	C2	Sam	19	Science	Dr. Jones	C

Primary Key: {student-id, course-id} (Composite Key)

Identifying the Problems:

- {student_id, course_id} → grade ✓ Full dependency (need both to determine grade)
- student_id → student_name ✗ Partial dependency (only need student_id)
- student_id → student_age ✗ Partial dependency (only need student_id)
- course_id → course_name ✗ Partial dependency (only need course_id)
- course_id → instructor ✗ Partial dependency (only need course_id)

Problems with this Design:

1. Data Redundancy:
   - "John" and "20" are repeated for every course John takes
   - "Math" and "Dr. Smith" are repeated for every student in Math
   - Storage space is wasted

2. Update Anomaly:

```
-- If John changes his name to "Jonathan":
-- Need to update multiple rows, might miss some
UPDATE course_enrollment SET student_name = 'Jonathan'
WHERE student_id = 1;
-- What if we accidentally miss one row? Inconsistent data!
```

3. Insert Anomaly:

```
-- Cannot add a new course without enrolling a student:
INSERT INTO course_enrollment (course_id, course_name, instructor)
VALUES ('C4', 'Physics', 'Dr. Wilson'); -- ERROR! Missing student_id
```

4. Delete Anomaly:

```
-- If Sam drops Science (only student in that course):
DELETE FROM course_enrollment WHERE student_id = 3 AND course_id = 'C2';
-- We lose all information about the science course!
```

How to Convert to 2NF (Good Design)

solution: Split into Multiple Tables

Students Table:

student_id	student_name	student_age
1	John	20
2	Mary	21
3	Sam	19

Courses Table:

course_id	course_name	instructor
C1	Math	Dr. Smith
C2	Science	Dr. Jones
C3	History	Dr. Brown

Enrollments Table:

student_id	course_id	grade
1	C1	A
1	C2	B
2	C1	B
2	C3	A
3	C2	C

Benefits of 2NF Design:

1. Eliminated Redundancy:
   - Each student's information stored only once
   - Each course's information stored only once
   - Only enrollment-specific data (grades) in enrollment table.

2. Easy Updates:

```
-- change John's name - only one place to update:
UPDATE students SET student_name = 'Jonathan' WHERE student_id = 1;

-- change Math instructor - only one place to update:
UPDATE courses SET instructor = 'Dr. Johnson' WHERE course_id = 'C1';
```

3. Easy Inserts:

```
-- Add new course without students:
INSERT INTO courses VALUES ('C4', 'Physics', 'Dr. Wilson');

-- Add new student without enrollments:
INSERT INTO students VALUES (4, 'Lisa', 22);
```

4. Safe Deletes:

```
-- Remove Sam's Science enrollment:
DELETE FROM enrollments WHERE student_id = 3 AND course_id = 'C2';
-- Course information is preserved in course table!
```

5. Better Queries:

```
--Get all courses John is taking:
SELECT c.course_name
FROM students s
JOIN enrollmentse ON s.student_id = e.student_id
JOIN courses c ON e.course_id = c.course_id
WHERE s.student_name = 'John';
```

Third Normal Form (3NF)

A table is in 3NF if:

1. It is already in 2NF
2. It has no transitive dependencies i.e. NO non-prime attribute is transitively dependent on the primary key.

Transitive Dependency: When a non-prime attribute depends on another non-prime attribute (instead of depending directly on the primary key). Transitive Dependency occurs when: $A \rightarrow B \rightarrow C$ (where A is primary key, B and C are non-prime attributes). This means C depends on A through B, not directly on A.

Pattern: Primary Key → Non-Prime Attribute → Another Non-Prime Attribute

Every non-key column should depend directly on the primary key, not on other non-key columns.

Example of 3NF Violation (Bad Design):

Employee Table (NOT in 3NF):

emp_id	emp_name	dept_id	dept_name	dept_location	emp_salary
101	John	D1	Sales	Building A	50000
102	Mary	D1	Sales	Building A	55000
103	Sam	D2	IT	Building B	60000
104	Lisa	D2	IT	Building B	65000
105	Tom	D3	HR	Building C	45000

Primary Key: emp_id

Identifying the Problems:

→ Functional Dependencies:

- emp_id → emp_name ✓ Direct dependency (employee ID determines name)
- emp_id → dept_id ✓ Direct dependency (employee ID determines department ID)
- emp_id → emp_salary ✓ Direct dependency (employee ID determines salary)
- dept_id → dept_name ✗ Causes transitive dependency
- dept_id → dept_location ✗ Causes transitive dependency

→ Transitive Dependencies:

- emp_id → dept_id → dept_name (emp_id determines dept_name indirectly through dept_id)
- emp_id → dept_id → dept_location (emp_id determines dept_location indirectly through dept_id)

Problems with this design:

1. Data Redundancy:
   - "Sales" and "Building A" are repeated for every Sales employee
   - "IT" and "Building B" are repeated for every IT employee
   - Department information is duplicated across multiple employee records

2. Update Anomaly:

```
-- If Sales department moves to Building D:
-- Need to update multiple employee records
UPDATE employee SET dept_location = 'Building D' WHERE dept_id = 'D1';
-- But what if someone accidentally updates only some rows?
```

UPDATE employee SET dept_location = 'Building D' WHERE emp_id = 101;
--Missed others!

--Now we have inconsistent data: some Sales employee show Building A, others Building D

3. Insert Anomaly:
-- Cannot add a new department without hiring an employee:
-- This won't work because emp_id is required (primary key)
INSERT INTO Employee (dept_id, dept_name, dept_location)
VALUES ('D4', 'Marketing', 'Building E'); --ERROR!

4. Delete Anomaly:
-- If Tom quits (only HR employee):
DELETE FROM employee WHERE emp_id = 105;
-- We lose all information about HR department (dept_name, dept_location)!!

5. Data Inconsistency Risk:
-- Someone might accidently enter wrong department info:
INSERT INTO employee VALUES (106, 'Alice', 'D1', 'Marketing', 'Building A', 52000);
-- Now D1 is associated with both "Sales" and "Marketing" - which is correct?

How to convert to 3NF (Good Design):
Solution: Remove Transitive Dependencies by Creating Separated Tables

Departments Table:

dept_id	dept_name	dept_location
D1	Sales	Building A
D2	IT	Building B
D3	HR	Building C

Employees Table:

emp_id	emp_name	dept_id	emp_salary
101	John	D1	50000
102	Mary	D1	55000
103	Sam	D2	60000
104	Lisa	D2	65000
105	Tom	D3	45000

Foreign Key: dept_id in Employees table references dept_id in Departments table.

Benefits of 3NF Design:

1. Eliminated Redundancy:
   - Each department's information stored only once in Department table
   - No duplication of department names or locations
   - Significant space saving in large

2. Consistent updates:

```
-- Move sales department to Building D - only one update needed:
UPDATE departments SET dept_location = 'Building D' WHERE dept_id = 'D1';
-- All sales employees automatically reflect the change when joined
```

3. Easy Department Management:

```
-- Add new department without employees:
INSERT INTO departments VALUES ('D4', 'Marketing', 'Building E');

-- Add new employee to existing department:
INSERT INTO employees VALUES (106, 'Alice', 'D1', 52000);
                                        -- Alice joins Sales
```

4. Safe Employee Operations :

```
-- Remove Tom (HR employee) :
DELETE FROM employees WHERE emp_id = 105;
-- HR department information is preserved in departments table !

-- Get all employees with their department info :
SELECT e.emp_name, e.emp_salary, d.dept_name, d.dept_location
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
```

5. Data Integrity :

- Foreign key constaint ensures employees can only be assigned to existing departments.
- Cannot accidentally create inconsistent department information.
- Referential integrity is maintained automatically.

6. Better Performance :

- Smaller employee table (no repeated department info)
- Faster queries on employee data
- Efficient joins when department information is needed.

Real-life Benefits :

- Scalability
- Maintenance
- Storage Efficiency
- Data consistency

Normalisation Part - 3

Design a database for a quora like app:

- User should be able to post a question
- User should be able to answer a question
- User should be able to comment on an answer
- User should be able to comment on a comment
- User should be able to like a comment or question or an answer
- User should be able to follow another user
- Every question can belong to multiple topics
- User can follow a topic also
- You should be able to filter out questions based on topic.

Soution:

1. Users Table - stores all user account information

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT, -- Unique user identifier
    username VARCHAR(50) UNIQUE NOT NULL, -- Unique username for login
    email VARCHAR(100) UNIQUE NOT NULL, -- Login email (must be unique)
    password_hash VARCHAR(255) NOT NULL, -- Encrypted password
    full_name VARCHAR(100), -- User's real name (optional)
    bio TEXT, -- User profile description
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- When account was created
);
```

2. Topics table - stores categories/subjects for questions

```
CREATE TABLE topics (
    id INT PRIMARY KEY AUTO_INCREMENT, -- Unique topic identifier
    name VARCHAR(100) UNIQUE NOT NULL, -- Topic name (e.g. "Technology", "Sports")
    description TEXT, -- Brief topic description
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- When topic was created
);
```

3. Questions table - Stores all user questions

```
CREATE TABLE questions (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique question identifier
user_id INT NOT NULL, -- Who posted the question
title VARCHAR(255) NOT NULL, -- Question title/ headline
content TEXT, -- Detailed question content (optional)
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- When question was posted
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP, -- Last edit time
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
                                    -- delete question if user deleted
);
```

4. Answers Table - stores all answers to questions

```
CREATE TABLE answers (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique answer identifier
question_id INT NOT NULL, -- Which question this answer
user_id INT NOT NULL, -- Who wrote the answer
content TEXT NOT NULL, -- Answer content
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- When answer was posted
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP, Last edit time
FOREIGN KEY (question_id) REFERENCES questions(id) ON DELETE
CASCADE, -- Delete answer if question deleted
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE
CASCADE -- Delete answer if user deleted
);
```

5. Comments table - handles comments on answers AND replies to other comments

```
CREATE TABLE comments (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique comment identifier
user_id INT NOT NULL, -- Who wrote the comment
answer_id INT NULL, -- If commenting on an answer (use this)
parent_comment_id INT NULL, -- If replying to another comment (use this)
content TEXT NOT NULL, -- Comment content
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, When comment was posted
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP, -- last edit time
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
FOREIGN KEY (answer_id) REFERENCES answers(id) ON DELETE CASCADE,
FOREIGN KEY (parent_comment_id) REFERENCES comments (id) ON
DELETE CASCADE,
--Either comment on answer OR reply to another comment, not both
CHECK ((answer_id IS NOT NULL AND parent_comment_id IS NULL) OR
        (answer_id IS NULL AND parent_comment_id IS NOT NULL))
);
```

6. Likes Table - handles likes for questions, answers, and comments in one table

```
CREATE TABLE likes (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique like identifier
user_id INT NOT NULL, -- Who gave the like
question_id INT NULL, -- If liking a question (use this)
answer_id INT NULL, -- If liking an answer (use this)
comment_id INT NULL, -- If liking a comment (use this)
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- When like was given
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
FOREIGN KEY (question_id) REFERENCES questions(id) ON DELETE CASCADE,
FOREIGN KEY (answer_id) REFERENCES answers(id) ON DELETE CASCADE,
FOREIGN KEY (comment_id) REFERENCES comments(id) ON DELETE CASCADE,
```

-- Ensure like is for exactly one type of content (question OR answer OR comment)
CHECK ((question_id IS NOT NULL AND answer_id IS NULL AND comment_id IS NULL) OR
(question_id IS NULL AND answer_id IS NOT NULL AND comment_id IS NULL) OR
(question_id IS NULL AND answer_id IS NULL AND comment_id IS NOT NULL));
-- Prevent duplicate likes from same user on same content
UNIQUE KEY unique_question_like (user_id, question_id),
UNIQUE KEY unique_answer_like (user_id, answer_id),
UNIQUE KEY unique_comment_like (user_id, comment_id),
);

7. Question-Topic mapping - links questions to topics (many-to-many relationship)
CREATE TABLE question_topics (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique mapping identifier
question_id INT NOT NULL, -- which question
topic_id INT NOT NULL, -- which topic it belongs to
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- when topic was added to question
FOREIGN KEY (question_id) REFERENCES questions(id) ON DELETE CASCADE,
FOREIGN KEY (topic_id) REFERENCES topics(id) ON DELETE CASCADE,
UNIQUE KEY unique_question_topic (question_id, topic_id) -- Prevent duplicate topic assignments
);

8. User follows - tracks when users follows other users
CREATE TABLE user_follows (
id INT PRIMARY KEY AUTO_INCREMENT, -- Unique follow relationship identifier
follower_id INT NOT NULL, -- user who is following (the follower)
following_id INT NOT NULL, -- user being followed (the one followed)
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- when follow relationship started

```
FOREIGN KEY (follower_id) REFERENCES users(id) ON DELETE CASCADE,
FOREIGN KEY (following_id) REFERENCES users(id) ON DELETE CASCADE,
-- Prevent users from following themselves
CHECK (follower_id ≠ following_id)
-- Prevent duplicate follows (same user following same person twice)
UNIQUE KEY unique_follow (follower_id, following_id)
);
```

9. Topic follows - tracks when users follow specific topics

```
CREATE TABLE topic_follows (
  id INT PRIMARY KEY AUTO-INCREMENT, -- Unique topic follow identifier
  user_id INT NOT NULL, -- User who is following the topic
  topic_id INT NOT NULL, -- Topic being followed
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- When user started following topic
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  FOREIGN KEY (topic_id) REFERENCES users(id) ON DELETE CASCADE,
  -- Prevent duplicate topic follows (same user following same topic twice)
  UNIQUE KEY unique_topic_follow (user_id, topic_id)
);
```

Boyce - Codd Normal Form (BCNF)

BCNF is a stricter version of 3NF (Third Normal Form).

A table is in BCNF if:

1. It is already in 3NF
2. For every functional dependency $X \rightarrow Y$, X must be a super key.

That means: If one column (or group of columns) determines another, then that determining column must be able to uniquely identify every row in the table.

Simple Rule:

If column A determines column B ($A \rightarrow B$), then column A must be able to uniquely identify every row in the table.

In BCNF, only columns that uniquely identify rows should be allowed to determine other columns.

Analogy:

Think of a school where only teachers (Keys) can give grades to students. If a student (non-key) could give grades to other students, that would be wrong! In BCNF, only "teacher-like" columns (Keys) can determine other information.

Example of BCNF Violation (Bad Design):

Student-Advisor Table (3NF but NOT BCNF)

student_id	subject	advisor_name
1	Math	Dr. Smith
1	Science	Dr. Jones
2	Math	Dr. Smith
2	History	Dr. Brown
3	Math	Dr. Smith
3	Science	Dr. Jones

Primary key : {student_id, subject} (composite key)

Understanding the Problem :

Function Dependencies :

- {student_id, subject} → advisor_name ✓ Valid (Key determines non-key)
- advisor_name → subject ✗ BCNF violation! (non-key determines non-key)

Why is advisor_name → subject a Problem ?

- Dr. Smith always teaches Math (Dr. Smith determines Math)
- Dr. Jones always teaches Science (Dr. Jones determines Science)
- Dr. Brown always teaches History (Dr. Brown determines History)
- This means advisor_name (a non-key) is determining subject (another non-key)

Explanation: Imagine you know the teacher's name, you can automatically tell what subject they teach. This creates a dependency where a non-key column (advisor_name) controls another column (subject). In BCNF, only keys should have this power!

Problems with this design :

1. Update Anomaly :

```
-- If Dr. Smith starts teaching Physics instead of Math:
-- Need to update multiple rows where Dr. Smith appears
UPDATE student_advisor SET subject = 'Physics' WHERE advisor_name
= 'Dr. Smith';
-- But this affects multiple students! Very risky.
```

2. Insert Anomaly:

-- Cannot add "Dr. Wilson teaches Chemistry" without assigning a student:
-- This won't work because we need student_id (part of primary key):

```
INSERT INTO student_advisor (subject, advisor_name) VALUES
('Chemistry', 'Dr. Wilson');
```

3. Delete Anomaly:

-- If student 3 drops all subjects:

```
DELETE FROM student_advisor WHERE student_id = 3;
```

-- We might lost the information that "Dr. Jones teaches Science" if no other students take Science.

4. Data Inconsistency:

-- Someone might accidentally enter:

```
INSERT INTO student_advisor VALUES (4, 'Math', 'Dr. Jones');
```

-- Now Dr. Jones teaches both Math and Science? Confusing!

How to Convert to BCNF (Good Design):

Solution: Separate the conflicting Dependency

Advisors Table:

advisor_name	subject
Dr. Smith	Math
Dr. Jones	Science
Dr. Jones	History

Student_Assignments Table:

student_id	advisor_name
1	Dr. Smith
1	Dr. Jones
2	Dr. Smith
2	Dr. Brown
3	Dr. Smith
3	Dr. Jones

Benefits of BCNF Design:

1. Clean Dependencies
   - In Advisors Table: advisor_name → subject ✓ (advisor_name is the key)
   - In Student_Assignments: student_id → advisor_name ✓ (can have multiple rows per student)

2. Easy Updates:

```
-- Dr. Smith switches from Math to Physics:
UPDATE advisors SET subject = 'Physics' WHERE advisor_name
= 'Dr. Smith';
-- One update affects all students automatically!
```

3. Easy Management:

```
-- Add new advisor without students:
INSERT INTO advisors VALUES ('Dr. Wilson', 'Chemistry');

-- Assign student to advisor:
INSERT INTO student_assignments VALUES (4, 'Dr. Wilson');
```

4. Data Consistency:

- Each advisor teaches exactly one subject (enforced by table structure)
- Cannot accidentally assign advisor to wrong subject
- Clear separation of concerns

What is Multivalues Dependency (MVD) ?

When one attribute determines multiple independent set of values for other attributes.

When knowing one piece of information gives you multiple lists of other information that are completely independent of each other.

Symbol: $A \twoheadrightarrow B$ (A multi-determines B)

Pattern: $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ (where B and C are independent)

Analogy:

Think of a person who has multiple hobbies AND multiple favorite foods.

If you know the person name:

- You get a list of all their hobbies (Reading, Swimming, Painting)
- You get a list of all their favorite foods (Pizza, Ice-cream, Pasta)
- But hobbies and foods are completely independent - loving Pizza doesn't relate to liking reading!

Example: Person-Interests Table:

person_name	hobby	favorite_food
John	Reading	Pizza
John	Reading	Ice cream
John	Swimming	Pizza
John	Swimming	Ice cream
Mary	Dancing	Pasta
Mary	Dancing	Salad
Mary	singing	Pasta
Mary	singing	Salad

Understanding the MVD:

Dependencies:

- person-name ↠ hobby (John's name gives us his hobbies: Reading, Swimming)
- person-name ↠ favorite-food (John's name gives us his foods: Pizza, Ice-cream)

The Problem:

- John has 2 hobbies and 2 favorite foods
- To store this, we need 2 × 2 = 4 rows!
- If John adds 1 more hobby, we need 3 × 2 = 6 rows
- If John adds 1 more hobby, we need 2 × 3 = 6 rows
- The table grows very quickly.

Why it's independent:

- John's hobby "Reading" has nothing to do with his favorite food "Pizza"
- We could rearrange: John + Reading + Ice-cream, John + Swimming + Pizza
- The combinations are forced, not meaningful.

Problems with MVD:

1. Unnecessary combinations:
   - Creates meaningless combinations of independent data
   - John + Reading + Pizza vs John + Reading + Ice cream are both forced combinations.

2. Storage Explosion:
   - If person has 5 hobbies and 3 favorite foods = 15 rows selected.
   - Adding 1 hobby requires adding 3 more rows (one for each food)

3. Update complexity:

- To add a hobby for John, must create rows for every favourite food.
- To add a favourite food, must create rows for every hobby

4. Delete Problems:

- cannot remove one hobby - food combination without losing data.
- Deleting "John + Reading + Pizza" might accidentally remove hobby or food info.

Fourth Normal form (4NF):

A table is in 4NF if:

1. It is already in BCNF
2. It has no multivalued dependencies

A table should not store multiple independent lists of information about the same entity.

Example of 4NF Violation (Bad Design):

Employee_Skills_Languages Table (BCNF but NOT 4NF):

emp_id	skill	language
1	Java	English
1	Java	Spanish
1	Python	English
1	Python	Spanish
2	HTML	French
2	HTML	German
2	CSS	French
2	CSS	German

Understanding the Problem:

Multivalued Dependencies:

- emp_id ↠ skill (Employee 1 has skills: Java, Python)
- emp_id ↠ language (Employee 1 speaks: English, Spanish)

Why It's Bad:

- Employee 1 has 2 skills and 2 language = 4 rows needed
- Skills and languages are completely independent
- Java doesn't relate to English any more than Java relates to Spanish
- We're forced to create meaningless combinations.

Problems with This Design:

1. Forced combination:
   - "Employee 1 + Java + English" vs "Employee 1 + Java + Spanish"
   - These combinations don't mean anything special - they're just required by the table structure.

2. Storage Waste:
   Employee with 3 skills and 4 languages = 3 x 4 = 12 rows!
   Employee with 5 skills and 2 languages = 5 x 2 = 10 rows!

3. Maintenance Nightmare
   -- To add "German" language for Employee 1:
   -- Must add one row for each existing skill:
   INSERT INTO emp_skills_language VALUES (1, 'Java', 'German');
   INSERT INTO emp_skills_language VALUES (1, 'Python', 'German');
   -- If employee has 10 skills, need 10 INSERT statements!

4. Inconsistency Risk :

-- Might accidentally miss adding a combination :

```
INSERT INTO emp-skills-languages VALUES (1, 'Java', 'German');
-- Forget to add (1, 'Python', 'German')
-- Now it looks like Employee 1 only uses German with Java
```

How to convert to 4NF (Good Design):

Solution: Split independent MVDs into separate tables :

Employee_skills Table:

emp_id	skill
1	Java
1	Python
2	HTML
2	CSS

Employee-Language Table

emp_id	language
1	English
1	Spanish
2	French
2	German

Benefits of 4NF Design:

1. Natural Storage:
   - Employee 1's skills stored once each
   - Employee 1's language stored once each
   - No forced combination!

2. Easy Maintenance :

```
-- Add new skill for Employee 1:
INSERT INTO employee_skills VALUES (1, 'Javascript'); -- Just one row!

-- Add new language for Employee 1:
INSERT INTO employee_language VALUES (1, 'French'); -- Just one row!
```

3. Accurate Queries:

```
-- Get all skills for Employee 1:
SELECT skill FROM employee_skills WHERE emp_id = 1;

-- Get all language for Employee 1:
SELECT language FROM employee_languages WHERE emp_id = 1;

-- Get employees who know both Java AND speak English:
SELECT es.emp_id
FROM employee_skills es
JOIN employee_languages el ON es.emp_id = el.emp_id
WHERE es.skill = 'Java' AND el.language = 'English';
```

4. Storage Efficiency:

- employee with 3 skills and 4 languages: 3 + 4 = 7 rows total
- Previously needed: 3 × 4 = 12 rows
- Significant space saving!

Mapping concepts:

Mapping concepts shows how many items in one group relate to items in another group. Think of it like counting relationships.

Types of Mappings:

1. One-to-One (1:1) Mapping: One entity relates to exactly one other entity.

Example: Person ⟷ Passport

- One person has exactly one passport
- One passport belongs to exactly one person

Person	PassportNumber
Alice	P123456
Bob	P789012

2. One-to-Many (1:M) Mapping : one entity relates to multiple other entities

Example: Mother ↔ children

- One mother can have many children.
- Each child has exactly one biological mother.

Mother	child
Sarah	Tommy
Sarah	Emma
Sarah	Jack
Lisa	Sophie

3. Many-to-Many (M:M) Mapping : Multiple entities relate to multiple other entities.

Example : Students ↔ Courses

- One student can take many courses
- One course can have many students

Student	course
John	Math
John	Science
Mary	Math
Mary	English
Tom	Science

Generalization & Specialization

→ Generalization :

combining multiple similar entities into one general entity by identifying their common characteristic.

Example :

Before Generalization:

- Teachers : {name, employee-id, subject, salary}
- Janitors : {name, employee-id, building, salary}
- Nurses : {name, employee-id, department, salary}

After Generalization :

- Employee : {name, employee-id, salary} ← General entity
- Teachers : {subject} ← specific attributes
- Janitors : {building} ← specific attributes
- Nurses : {department} ← specific attributes

Database Design :

Employees Table :

emp-id	name	salary
1	John	50000
2	Mary	45000
3	Sam	55000
4	Lisa	40000
5	Tina	60000

Teachers Table :

emp-id	subject
1	Math
3	Science

Janitors Table :

emp-id	building
2	A
4	B

Nurses Table :

emp-id	department
5	Emergency

→ Specialization:

Breaking down a general entity into more specific entities based on their unique characteristic.

Example:

Before Specialization:

- Vehicles : {vehicle_id, brand, price, type, num_wheels, num_doors, wingspand}

After Specialization:

- Vehicles : {vehicle_id, brand, price} ← General Attribute
- Cars : {num_doors, num_wheels} ← Car specific
- Motorcycles : {num_wheels} ← Motorcycle - specific
- Airplanes : {wingspand} ← Airplane - specific

Database Design:

Vehicles Table

Vehicle_id	brand
1	Toyota
2	Honda
3	BMW
4	Yamaha
5	Boeing

Cars Table:

Vehicle_id	doors
1	4
3	4

Motorcycles Table:

vehicle_id	Wheels
2	2
4	2

Airplanes Table

Vehicle_id	wingspan
5	60.9

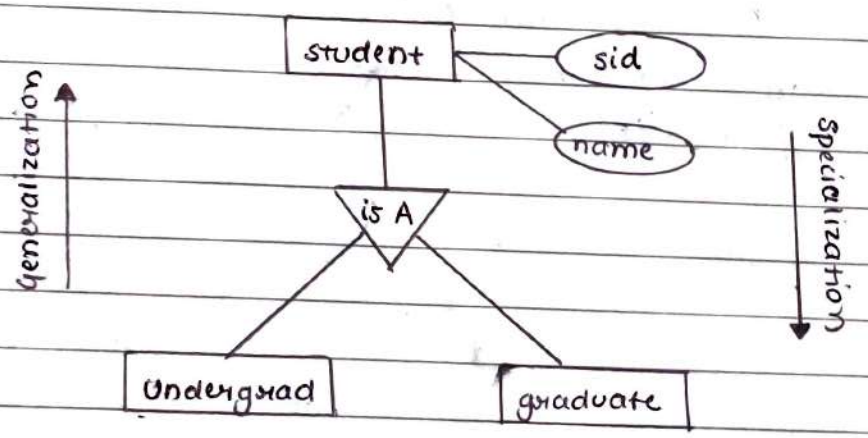
Benefits :

Generalization Benefits :

- Reduces redundancy (common attributes stored once)
- Easier to add new employee types
- consistent handling of common operations (like salary calculation)

Specialization

- Each entity type has only relevant attributes
- No null values for inapplicable attributes (cars don't have wingspan)
- Better performance (smaller, focused tables)



Through Table (Join Table)

A through table is a separate table created to handle many-to-many relationships between two other tables.

A through table (also called junction table or join table) is like a bridge that connects two other tables in a many-to many relationship.

Why use it?

You cannot directly connect two tables in a many-to-many relationship without creating a mess.

Example:

Imagine you have:

- Students who can take multiple courses
- Courses that can have multiple students

You cannot put this in one table without creating a mess!

→ Wrong way (without Through table):

Student table:

Student_ID	Name	Courses
1	Alice	Math, Science
2	Bob	Math, English

Problems:

Hard to search, update, or manage

→ Right way (with Through Table):

Students Table:

Student_ID	Name
1	Alice
2	Bob

Courses Table

Course_ID	Course_Name
101	Math
102	Science
103	English

Enrollments Table (Through table):

Student_ID	Course_ID	Grade	Semester
1	101	A	Fall2024
1	102	B+	Fall2024
2	101	B	Fall2024
2	103	A-	Fall2024

Benefits:

- Clean data organization
- Easy to add/remove relationships
- Can store additional relationship information.
- Efficient queries
- No data duplication