Normalization:

Normalization is a core database design principle, not specific to MySQL, used to organize data efficiently to reduce redundancy and improve data integrity. This is achieved by systematically breaking down large tables into smaller, related tables and defining relationships between them using keys (primary and foreign).

The primary goals of normalization are to:

Eliminate redundant data: Storing each piece of information only once saves storage space and prevents inconsistencies.

Prevent anomalies: It helps avoid insertion, update, and deletion anomalies that can lead to data corruption.

Enhance data integrity: By enforcing rules and constraints, normalization ensures the accuracy and consistency of data.

Simplify database maintenance and improve scalability.

Normalization is achieved through a series of "Normal Forms" (NF), each with specific rules that build upon the last. The most common are:

First Normal Form (1NF): The table must have atomic (indivisible) values in each column, and there should be no repeating groups of data. Each row must be unique, typically identified by a primary key.

Second Normal Form (2NF): The table must be in 1NF, and all non-key attributes must be fully functionally dependent on the entire primary key. This means no non-key attribute should depend on only a part of a composite primary key.

Third Normal Form (3NF): The table must be in 2NF, and there should be no transitive dependencies. In simple terms, non-key attributes should not be dependent on other non-key attributes

First Normal Form (1NF)

Rule: Each column must contain only a single, atomic (indivisible) value, and there should be no repeating groups of data.

Problem Example (Unnormalized users table):

userid	fullname	phone_numbers
1	John Doe	123-4567, 890-1234
2	Jane Smith	555-6789

Violation: The fullname column can be split into two (first/last name), and phone_numbers contains multiple values.

After Normalization:

userid	firstname	lastname
1	John	Doe
2	Jane	Smith

Second Normal Form (2NF)

Rule: A table must be in 1NF, and all non-key attributes must be fully dependent on the entire primary key. This typically applies to tables with a composite primary key.

Problem Example (student_projects table):

Primary Key: (StudentID, ProjectNo).

StudentID ProjectNo StudentName ProjectName

S001 P1 Alice Database Design

S001 P2 Alice MySQL Basics

S002 P1 Bob Database Design

Violation: StudentName depends only on StudentID, not the whole key (StudentID, ProjectNo). This is a partial dependency and causes redundancy (Alice's name is repeated).

Normalized Example (2NF):

We separate attributes that only depend on a part of the composite key into new tables.

students table:

StudentID StudentName

S001 Alice

S002 Bob

projects table:

ProjectNo ProjectName

P1 Database Design

P2 MySQL Basics

student_project_enrollment table (link table):

StudentID	ProjectNo
S001	P1
S001	P2
S002	P1

Third Normal Form (3NF)

Rule: A table must be in 2NF, and there should be no transitive dependencies. This means non-key attributes should not depend on other non-key attributes.

Problem Example (employees table):

Primary Key: EmployeeID.

Dependency chain: EmployeeID → JobCode → JobTitle.

EmployeeID	Name	JobCode	JobTitle
E001	Alice	J01	Chef
E002	Bob	J02	Waiter
E003	Charlie	J01	Chef

Violation: JobTitle depends on JobCode, which in turn depends on the primary key EmployeeID. The job title "Chef" is repeated.

Normalized Example (3NF):

We move the transitively dependent columns into a new table.

employees table:

EmployeeID	Name	JobCode
E001	Alice	J01
E002	Bob	J02
E003	Charlie	J01

jobs table:

JobCode	JobTitle
J01	Chef
J02	Waiter

Types of Joins

INNER JOIN: Returns rows with matches in both tables.

LEFT JOIN: Returns all rows from the left table and matching rows from the right table, using NULL for non-matches.

RIGHT JOIN: Returns all rows from the right table and matching rows from the left table, using NULL for non-matches.

FULL OUTER JOIN (Simulated): MySQL does not have a native FULL OUTER JOIN but it can be simulated using LEFT JOIN and RIGHT JOIN with UNION.

CROSS JOIN: Creates a Cartesian product of all rows from both tables.

SELF JOIN: Joins a table to itself using aliases

```
CREATE TABLE employees (
   employee_id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100),
   department_id INT
);

CREATE TABLE departments (
   department_id INT AUTO_INCREMENT PRIMARY KEY,
   department_name VARCHAR(100)
);
```

Inserting Values:

```
INSERT INTO employees (name, department_id) VALUES
('Alice', 1),
('Bob', 2),
('Charlie', 1),
('David', 3),
('Eve', NULL);
```

```
INSERT INTO departments (department_id, department_name) VALUES
(1, 'HR'),
(2, 'Engineering'),
(3, 'Marketing'),
(4, 'Finance');
```

Inner Join:

```
SELECT column_names
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

Left Join:

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

Right Join:

```
SELECT employees.name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

Full Join:

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id
UNION
```

```
SELECT employees.name, departments.department_name

FROM employees

RIGHT JOIN departments

ON employees.department_id = departments.department_id;
```

Cross Join:

```
SELECT employees.name, departments.department_name

FROM employees

CROSS JOIN departments;
```

SELF JOIN

A self join is a type of join in which a table is joined to itself. This is useful when you need to compare rows within the same table, such as relating employees to their managers in an organizational hierarchy.

Consider an employees table that needs to represent employees and their managers. To achieve this, we can introduce a new column called manager_id in the employees table to store the ID of each employee's manager. After updating the table, we'll perform a self join to associate each employee with their manager.

Step 1: Altering the Table to Add the manager_id Column

```
ALTER TABLE employees ADD COLUMN manager_id INT;
```

Step 2: Updating the Table with Manager Information

```
UPDATE employees SET manager_id = 3 WHERE employee_id = 1;

UPDATE employees SET manager_id = 3 WHERE employee_id = 2;

UPDATE employees SET manager_id = 4 WHERE employee_id = 3;
```

Query:

```
SELECT a.name AS employee, b.name AS manager

FROM employees a, employees b

WHERE a.manager_id = b.employee_id;
```