**Task 5**

1. **Difference between INNER and LEFT JOIN?**

The difference between **INNER JOIN** and **LEFT JOIN** in SQL comes down to **which rows are included** in the result set when there is **no match** between the joined tables.

**🔸 INNER JOIN**

* **Returns only the rows where there is a match in both tables.**
* If there is no match, the row is **excluded** from the result.

### 🔸 LEFT JOIN (or LEFT OUTER JOIN)

* **Returns all rows from the left table**, and the **matching rows** from the right table.
* If there is **no match** on the right side, the result will contain **NULLs** for the right table’s columns.

Let's walk through a practical example with **sample data** to show exactly how INNER JOIN and LEFT JOIN behave.

## **🎯 Tables:**

## Table 1: customers

| **customer\_id** | **name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |
| 3 | Charlie |

### Table 2: orders

| **order\_id** | **customer\_id** | **amount** |
| --- | --- | --- |
| 101 | 1 | 50 |
| 102 | 1 | 75 |
| 103 | 2 | 100 |

* 📝 Note: Charlie (customer\_id = 3) has **no orders**.

## 🔹 INNER JOIN Example

SELECT c.customer\_id, c.name, o.order\_id, o.amount

FROM customers c

INNER JOIN orders o ON c.customer\_id = o.customer\_id;

### 🔍 Result:

| **customer\_id** | **name** | **order\_id** | **amount** |
| --- | --- | --- | --- |
| 1 | Alice | 101 | 50 |
| 1 | Alice | 102 | 75 |
| 2 | Bob | 103 | 100 |

✅ Only customers who **have orders** are shown.

## 🔸 LEFT JOIN Example

SELECT c.customer\_id, c.name, o.order\_id, o.amount

FROM customers c

LEFT JOIN orders o ON c.customer\_id = o.customer\_id;

### 🔍 Result:

| **customer\_id** | **name** | **order\_id** | **amount** |
| --- | --- | --- | --- |
| 1 | Alice | 101 | 50 |
| 1 | Alice | 102 | 75 |
| 2 | Bob | 103 | 100 |
| 3 | Charlie | NULL | NULL |

✅ All customers are shown.  
❌ For Charlie (no orders), the order\_id and amount fields are **NULL**.

1. **What is a FULL OUTER JOIN?**

A **FULL OUTER JOIN** is a type of SQL join that returns **all records** when there is a match in **either** the left or right table. If there is **no match**, the result will still include the row, but with NULL values in the columns where the match is missing.

### Syntax:

SELECT \*

FROM table1

FULL OUTER JOIN table2

ON table1.id = table2.id;

### What it does:

* Combines the results of both **LEFT JOIN** and **RIGHT JOIN**.
* If a row from table1 has no matching row in table2, it will still appear with NULLs for table2 columns.
* If a row from table2 has no matching row in table1, it will still appear with NULLs for table1 columns.

### Example:

#### Table: Customers

| **CustomerID** | **Name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |
| 3 | Carol |

#### Table: Orders

| **OrderID** | **CustomerID** |
| --- | --- |
| 101 | 1 |
| 102 | 4 |

#### Query:

SELECT \*

FROM Customers

FULL OUTER JOIN Orders

ON Customers.CustomerID = Orders.CustomerID;

#### Result:

| **CustomerID** | **Name** | **OrderID** | **CustomerID** |
| --- | --- | --- | --- |
| 1 | Alice | 101 | 1 |
| 2 | Bob | NULL | NULL |
| 3 | Carol | NULL | NULL |
| NULL | NULL | 102 | 4 |

This shows:

* Alice has an order → row with full data.
* Bob and Carol have no orders → show NULLs in order columns.
* Order 102 has no matching customer → show NULLs in customer columns.

1. **Can joins be nested?**

Yes, joins can be nested in SQL.

✅ What does “nested joins” mean?

It means using one join inside another, or chaining multiple joins together in a query. You can:

* Join two tables, and then join the result to a third table.
* Use parentheses to control the order of operations (especially in complex joins).
* Combine different types of joins (e.g., INNER JOIN inside an OUTER JOIN).

🔍 Example 1: Simple Nested JOIN

SELECT \*

FROM (Customers

INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID)

INNER JOIN Products ON Orders.ProductID = Products.ProductID;

Here:

1. Customers and Orders are joined first.
2. That result is then joined with Products.

🔍 Example 2: Mixed Join Types

SELECT \*

FROM (Employees

LEFT JOIN Departments ON Employees.DepartmentID = Departments.ID)

FULL OUTER JOIN Projects ON Employees.ProjectID = Projects.ID;

* First, a LEFT JOIN between Employees and Departments.
* Then a FULL OUTER JOIN with Projects.

📌 Notes:

* SQL joins are associative, so the order of joining generally doesn’t affect the final result unless:
  + You're using LEFT or RIGHT OUTER JOIN (which are directional).
  + You're adding WHERE, GROUP BY, or filtering that depends on join results.
* Parentheses help organize complex logic and ensure clarity.

1. **How to join more than 2 tables?**

To **join more than 2 tables** in SQL, you simply **chain multiple JOIN clauses** together. Each join connects one more table to the growing result set, based on a common key (usually a foreign key relationship).

### ✅ General Syntax:

*SELECT columns*

*FROM table1*

*JOIN table2 ON table1.column = table2.column*

*JOIN table3 ON table2.column = table3.column*

*... and so on*;

You can use any type of join (INNER, LEFT, RIGHT, FULL, etc.) depending on your need.

### 🔍 Example: Joining 3 Tables

Let’s say you have these tables:

#### 1. Customers

| **CustomerID** | **Name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |

#### 2. Orders

| **OrderID** | **CustomerID** | **ProductID** |
| --- | --- | --- |
| 101 | 1 | 1001 |
| 102 | 2 | 1002 |

#### 3. Products

| **ProductID** | **ProductName** |
| --- | --- |
| 1001 | Laptop |
| 1002 | Phone |

#### 🔗 SQL Query:

SELECT Customers.Name, Orders.OrderID, Products.ProductName

FROM Customers

JOIN Orders ON Customers.CustomerID = Orders.CustomerID

JOIN Products ON Orders.ProductID = Products.ProductID;

#### ✅ Result:

| **Name** | **OrderID** | **ProductName** |
| --- | --- | --- |
| Alice | 101 | Laptop |
| Bob | 102 | Phone |

### 🧠 Tips:

* Make sure each JOIN has a proper **ON** condition to avoid a **Cartesian product**.
* Use **table aliases** (C, O, P) to make complex queries shorter and clearer.
* Be mindful of NULL values if using LEFT or RIGHT JOIN.

1. **What is a cross join?**

A **CROSS JOIN** is a type of SQL join that returns the **Cartesian product** of two tables — that means **every row from the first table is paired with every row from the second table**, regardless of any condition.

### 🔹 Syntax:

SELECT \*

FROM table1

CROSS JOIN table2;

Or (in some databases):

SELECT \*

FROM table1, table2;

### 🔍 Example:

#### Table A: Colors

| **ColorID** | **Color** |
| --- | --- |
| 1 | Red |
| 2 | Blue |

#### Table B: Sizes

| **SizeID** | **Size** |
| --- | --- |
| 1 | Small |
| 2 | Medium |
| 3 | Large |

#### Query:

SELECT \*

FROM Colors

CROSS JOIN Sizes;

#### 🔸 Result (2 × 3 = 6 rows):

| **ColorID** | **Color** | **SizeID** | **Size** |
| --- | --- | --- | --- |
| 1 | Red | 1 | Small |
| 1 | Red | 2 | Medium |
| 1 | Red | 3 | Large |
| 2 | Blue | 1 | Small |
| 2 | Blue | 2 | Medium |
| 2 | Blue | 3 | Large |

### ✅ When to Use CROSS JOIN:

* To generate combinations (like all possible color/size pairs).
* In data science for parameter grid creation.
* In testing or demo datasets.

1. What is a natural join?

A **NATURAL JOIN** is a type of SQL join that **automatically matches columns** between two tables based on **columns with the same name and compatible data types**, and performs an **INNER JOIN** on those columns.

### ✅ Key Characteristics of a NATURAL JOIN:

* **No ON clause needed** — the join condition is implicit.
* It matches **all columns with the same name** in both tables.
* Returns only **one column** for each pair of matching columns (i.e., duplicates are removed).
* If there are **no columns with the same name**, the result is a **Cartesian product**.

### 🔹 Syntax:

SELECT \*

FROM table1

NATURAL JOIN table2;

### 🔍 Example:

#### Table: Employees

| **EmpID** | **Name** | **DeptID** |
| --- | --- | --- |
| 1 | Alice | 10 |
| 2 | Bob | 20 |

#### Table: Departments

| **DeptID** | **DeptName** |
| --- | --- |
| 10 | HR |
| 20 | Engineering |

#### Query:

SELECT \*

FROM Employees

NATURAL JOIN Departments;

#### 🔸 Result:

| **EmpID** | **Name** | **DeptID** | **DeptName** |
| --- | --- | --- | --- |
| 1 | Alice | 10 | HR |
| 2 | Bob | 20 | Engineering |

Here, the join was done **automatically** on DeptID because it's the only column with the same name in both tables.

1. **Can you join tables without foreign key?**

**Yes, you can join tables without a foreign key.**

A **foreign key** is used to **enforce data integrity**, but it is **not required** to perform a **JOIN** in SQL.

**✅ Key Points:**

* A **JOIN** just needs **columns with related data**.
* The database doesn’t care if there’s a foreign key constraint or not — it only matches values during the JOIN.
* example to show how you can **join tables without a foreign key**.

### 🟩 ****Step 1: Two Tables (No Foreign Key)****

#### 🔹 Table: employees

| **employee\_id** | **name** | **department\_code** |
| --- | --- | --- |
| 1 | Alice | D01 |
| 2 | Bob | D02 |
| 3 | Carol | D03 |

#### 🔹 Table: departments

| **code** | **department\_name** |
| --- | --- |
| D01 | HR |
| D02 | IT |
| D03 | Finance |

📌 **Note:** There is **no foreign key** defined between employees.department\_code and departments.code.

### 🟦 ****Step 2: Join Query (Without Foreign Key)****

SELECT

e.name,

d.department\_name

FROM

employees e

JOIN

departments d

ON

e.department\_code = d.code;

### 🟨 ****Result of the JOIN****

| **name** | **department\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Carol | Finance |

✅ The join works perfectly **without any foreign key**, as long as the values match.

1. **What is a self-join?**

A **self-join** is a SQL join where a **table is joined to itself**.

It’s useful when you want to compare **rows within the same table** — for example, finding relationships like **employees and their managers**, or **products that are similar**.

### 🔄 ****Why Use a Self-Join?****

You use a self-join when:

* There’s a **hierarchical relationship** in the table (like manager/employee).
* You need to **compare rows** from the same table.
* You want to find **duplicates**, **matches**, or **associations** within the same dataset.

### 📌 ****Example: Employee & Manager Relationship****

#### 🟩 Table: employees

| **employee\_id** | **name** | **manager\_id** |
| --- | --- | --- |
| 1 | Alice | NULL |
| 2 | Bob | 1 |
| 3 | Charlie | 1 |
| 4 | Diana | 2 |

📌 manager\_id points to another employee\_id in the **same table**.

### 🔁 Self-Join Query

SELECT

e.name AS employee,

m.name AS manager

FROM

employees e

LEFT JOIN

employees m

ON

e.manager\_id = m.employee\_id;

### ✅ Result

| **employee** | **manager** |
| --- | --- |
| Alice | NULL |
| Bob | Alice |
| Charlie | Alice |
| Diana | Bob |

1. **What causes Cartesian product?**

A **Cartesian product** in SQL is caused when you perform a **JOIN without a proper condition** (i.e., **no** ON **clause** or an incorrect one).

### 📌 **Definition:**

A **Cartesian product** returns **all possible combinations** of rows from two or more tables.

* If **Table A** has m rows and **Table B** has n rows, a Cartesian product will return m × n rows.

### ❗ When Does It Happen?

1. **Missing** ON **clause in a JOIN**
2. Using **CROSS JOIN** explicitly
3. Mistakenly writing a JOIN without specifying how the tables are related

### 🧪 Example (with missing join condition)

#### Table: students

| id | name |
| --- | --- |
| 1 | Alice |
| 2 | Bob |

#### Table: courses

| id | course\_name |
| --- | --- |
| A | Math |
| B | Science |

### 🔴 Query that causes Cartesian product:

SELECT \*

FROM students, courses;

or:

SELECT \*

FROM students

JOIN courses;

-- No ON condition!

### 🧨 Result: Cartesian Product (2 × 2 = 4 rows)

| id | name | id | course\_name |
| --- | --- | --- | --- |
| 1 | Alice | A | Math |
| 1 | Alice | B | Science |
| 2 | Bob | A | Math |
| 2 | Bob | B | Science |

### ✅ How to Fix It (Use Proper JOIN Condition)

SELECT \*

FROM students

JOIN courses ON students.id = courses.id;

1. **How to optimize joins?**

Optimizing **JOINs** is crucial for improving SQL query performance, especially when working with large datasets. Here's a clear, practical guide:

## ✅ 10 Ways to Optimize JOINs in SQL

### 1. ****Use Proper Indexing****

* **Create indexes** on columns used in the JOIN, WHERE, and ON clauses.

**Example:**

CREATE INDEX idx\_orders\_customer\_id ON orders(customer\_id);

### 2. ****Join on Indexed and Primary Key Columns****

* Prefer joining on **primary keys** or **indexed foreign keys** — they’re faster to search and match.

### 3. ****Avoid Joining Unnecessary Columns****

* Use SELECT specific\_columns instead of SELECT \* to reduce data transfer and processing.

SELECT c.name, o.order\_date

FROM customers c

JOIN orders o ON c.id = o.customer\_id;

### 4. ****Filter Early with WHERE****

* Apply filters **before the join** to reduce the number of rows involved.

-- Better

SELECT ...

FROM orders o

JOIN customers c ON o.customer\_id = c.id

WHERE o.status = 'completed';

### 5. ****Choose the Right JOIN Type****

* Use the **simplest JOIN** that serves your purpose:
  + Use INNER JOIN instead of LEFT JOIN if you don’t need unmatched rows.

### 6. ****Use Aliases and Fully Qualify Columns****

* Improves readability and helps avoid mistakes in complex queries.

SELECT a.name, b.salary

FROM employees a

JOIN payroll b ON a.id = b.employee\_id;

### 7. ****Avoid Functions in JOIN Conditions****

* Avoid this (slow):

ON UPPER(a.email) = UPPER(b.email)

* Better:

ON a.email = b.email

-- (Assumes emails are stored in a consistent case)

### 8. ****Use EXISTS Instead of JOIN for Filtering****

* Sometimes EXISTS performs better than INNER JOIN, especially for filtering.

SELECT \*

FROM employees e

WHERE EXISTS (

SELECT 1 FROM departments d

WHERE d.manager\_id = e.id

);

### 9. ****Denormalize When Necessary****

* For read-heavy systems (e.g., reporting), it's sometimes better to **pre-join** and store data in summary tables.

### 10. ****Analyze with EXPLAIN / Execution Plan****

* Use tools like EXPLAIN (MySQL/PostgreSQL) or Query Plan (SQL Server) to see:
  + Which indexes are used
  + Join order
  + Cost of each operation

EXPLAIN SELECT ...