Step 1: Create the `employee` and `office` tables

Before applying the joins, we'll first create and insert records into the `employee` and `office` tables:

```
-- Create employee table
CREATE TABLE employee (
    employee_id INT PRIMARY KEY,    -- Unique identifier for each employee
    first_name VARCHAR(50),         -- Employee's first name
    last_name VARCHAR(50),          -- Employee's last name
    office_id INT                   -- The office where the employee works (linked to office table)
);

-- Create office table
CREATE TABLE office (
    office_id INT PRIMARY KEY,      -- Unique identifier for each office
    city VARCHAR(50),               -- The city where the office is located
    country VARCHAR(50)             -- The country where the office is located
);

-- Insert records into employee table
INSERT INTO employee (employee_id, first_name, last_name, office_id) VALUES
(1, 'John', 'Doe', 101),            -- Employee John Doe works in office 101
(2, 'Jane', 'Smith', 102),          -- Employee Jane Smith works in office 102
(3, 'Michael', 'Johnson', 101),     -- Employee Michael Johnson works in office 101
(4, 'Emily', 'Davis', 103),         -- Employee Emily Davis works in office 103
(5, 'Robert', 'Brown', NULL);       -- Robert Brown has no office assigned (NULL office_id)

-- Insert records into office table
INSERT INTO office (office_id, city, country) VALUES
(101, 'New York', 'USA'),           -- Office 101 is in New York, USA
(102, 'London', 'UK'),              -- Office 102 is in London, UK
(103, 'Berlin', 'Germany');         -- Office 103 is in Berlin, Germany
```

1. **INNER JOIN**

Returns records that have matching values in both tables.

```
SELECT e.first_name, e.last_name, o.city, o.country
FROM employee e
INNER JOIN office o ON e.office_id = o.office_id;
```

Explanation:

- `SELECT e.first_name, e.last_name, o.city, o.country`:
  - Selects four columns:
    - `e.first_name`: First name from the `employee` table.
    - `e.last_name`: Last name from the `employee` table.
    - `o.city`: City from the `office` table.
    - `o.country`: Country from the `office` table.
- `FROM employee e`:
  - Specifies that the `employee` table is being used as the main table (aliased as `e` for shorter reference).
- `INNER JOIN office o`:
  - Joins the `employee` table (`e`) with the `office` table (`o`).
  - `INNER JOIN` ensures that only rows with matching `office_id` values in both tables are included in the result.
- `ON e.office_id = o.office_id`:
  - Defines the condition for the join, where `office_id` in the `employee` table must match `office_id` in the `office` table.
  - If an `employee` has an `office_id` that exists in the `office` table, their information is included.

Result:

- Employees with an assigned office (John, Jane, Michael, Emily) will be shown with their office locations. Employees without an office (Robert) are excluded.

1. `LIKE` Condition:

- The `LIKE` condition is used in string pattern matching. It can be used in joins to match columns with similar patterns.

Example:

Suppose you have a column `first_name` in the `employee` table, and you want to join only those rows where the first name starts with 'J'.

SELECT e.first_name, e.last_name, o.city

FROM employee e

INNER JOIN office o ON e.office_id = o.office_id

WHERE e.first_name LIKE 'J%';

Explanation:

- `LIKE 'J%'`: This condition matches employees whose first name starts with the letter 'J'.
- The join returns only employees whose first name starts with 'J', along with their office details.

2. `IN` Condition:

- The `IN` condition is used to check if a column value exists within a specified list of values. It can also be used within joins.

Example:

If you want to select employees who work in specific offices (say, offices with `office_id` 101 and 103):

SELECT e.first_name, e.last_name, o.city

FROM employee e

INNER JOIN office o ON e.office_id = o.office_id

WHERE e.office_id IN (101, 103);

Explanation:

- `WHERE e.office_id IN (101, 103)`: This condition filters only those employees whose `office_id` is either 101 or 103.
- It combines with the `INNER JOIN` to return matching employees who work in New York (101) and Berlin (103).

3. **`IS NULL` / `IS NOT NULL` Condition**:

- Used to filter records where certain columns are either `NULL` or not `NULL`.

Example:

Suppose you want to join all employees and return only those who have not been assigned an office (`office_id` is `NULL`).\

SELECT e.first_name, e.last_name

FROM employee e

LEFT JOIN office o ON e.office_id = o.office_id

WHERE e.office_id IS NULL;

Explanation:

- `WHERE e.office_id IS NULL`: This condition filters only those employees who don't have an assigned `office_id` (i.e., the `office_id` is `NULL`).
- The `LEFT JOIN` ensures that all employees are included, and this condition filters for those with no office.

2. **LEFT JOIN (or LEFT OUTER JOIN):**

Returns all records from the left table (`employee`), and the matched records from the right table (`office`). If there is no match, `NULL` is returned for the right table columns**.**

SELECT e.first_name, e.last_name, o.city, o.country
FROM employee e
LEFT JOIN office o ON e.office_id = o.office_id;

Explanation:

- `SELECT e.first_name, e.last_name, o.city, o.country`:
  - Similar to the `INNER JOIN`, selects first name, last name, city, and country.
- `FROM employee e`:
  - The `employee` table is again the main table (aliased as `e`).
- `LEFT JOIN office o`:
  - Joins the `employee` table with the `office` table.
  - `LEFT JOIN` includes all rows from the `employee` table, regardless of whether there is a matching `office_id` in the `office` table.
- `ON e.office_id = o.office_id`:
  - The condition for the join. If the `employee` has no `office_id` (e.g., Robert Brown has `NULL`), `NULL` will be returned for the `city` and `country` from the `office` table.

Result:

- All employees (including Robert who has no office) will be shown. For employees without an office, `city` and `country` will be `NULL`.

2. Range Condition (`BETWEEN`):

- Used to filter records by a range of values.

```
SELECT e.first_name, e.last_name, o.city, o.country

FROM employee e

LEFT JOIN office o ON e.office_id = o.office_id

WHERE e.employee_id BETWEEN 2 AND 4;
```

3. **RIGHT JOIN (or RIGHT OUTER JOIN)**

Returns all records from the right table (`office`), and the matched records from the left table (`employee`). If there is no match, `NULL` is returned for the left table columns.

```
SELECT e.first_name, e.last_name, o.city, o.country
FROM employee e
RIGHT JOIN office o ON e.office_id = o.office_id;
```

Explanation:

- `SELECT e.first_name, e.last_name, o.city, o.country`:
  - Same selection of columns as before.
- `FROM employee e`:
  - The `employee` table is used as the left table (aliased as `e`).
- `RIGHT JOIN office o`:
  - Joins the `employee` table with the `office` table.
  - `RIGHT JOIN` includes all rows from the `office` table, even if there are no matching employees assigned to an office.
- `ON e.office_id = o.office_id`:
  - The join condition. If there is no matching `employee` for an `office_id`, the employee's columns (`first_name`, `last_name`) will be `NULL`.

Result:

- All offices will be listed. If an office has no employees, the employee's first name and last name will be `NULL`.

3. `IS NULL` / `IS NOT NULL` Condition:

- Used to filter records where certain columns are either `NULL` or not `NULL`.

Example:

Suppose you want to join all employees and return only those who have not been assigned an office (`office_id` is `NULL`).

SELECT e.first_name, e.last_name

FROM employee e

LEFT JOIN office o ON e.office_id = o.office_id

WHERE e.office_id IS NULL;

Explanation:

- `WHERE e.office_id IS NULL`: This condition filters only those employees who don't have an assigned `office_id` (i.e., the `office_id` is `NULL`).
- The `LEFT JOIN` ensures that all employees are included, and this condition filters for those with no office.

4. FULL OUTER JOIN (Simulated with `UNION`)

Returns all records when there is a match in either table. If there is no match, `NULL` is returned for the missing side. We use `UNION` to simulate a `FULL OUTER JOIN` in systems that don't support it directly

```
SELECT e.first_name, e.last_name, o.city, o.country
FROM employee e
LEFT JOIN office o ON e.office_id = o.office_id
UNION
SELECT e.first_name, e.last_name, o.city, o.country
```

```
FROM employee e
RIGHT JOIN office o ON e.office_id = o.office_id;
```

Explanation:

- **LEFT JOIN part**:
  - Retrieves all employees and their corresponding offices (if any). Employees without an office will have `NULL` for office information.
- **RIGHT JOIN part**:
  - Retrieves all offices and their corresponding employees (if any). Offices without employees will have `NULL` for employee information.
- `UNION`:
  - Combines the results from the `LEFT JOIN` and `RIGHT JOIN`. This way, all employees and all offices are included, even if they don't match.

Result:

- All employees and offices are included. If an employee has no office, the office details are `NULL`. If an office has no employees, the employee details are `NULL`.

5. **CROSS JOIN**

Produces the Cartesian product of the two tables. Every row from the `employee` table is combined with every row from the `office` table.

```
SELECT e.first_name, e.last_name, o.city, o.country
FROM employee e
CROSS JOIN office o;
```

Explanation:

- `CROSS JOIN` creates a combination of every employee with every office.
- There is no `ON` condition in a `CROSS JOIN`, as it simply pairs every row from `employee` with every row from `office`.

Result:

- If there are 5 employees and 3 offices, the result will have 15 rows (every employee combined with every office).

Self Join

Let's dive into the concept of a Self Join in SQL and how it works in detail. To make it clear, we will create an `employee` table and use a self join to find the manager of every employee.

Step 1: Create the `employee` Table

Here, we will define an `employee` table where each employee can have a `manager_id` (referring to another employee in the same table).

```
CREATE TABLE employee (
    employee_id INT PRIMARY KEY,   -- Unique ID for each employee
    first_name VARCHAR(50),        -- Employee's first name
    last_name VARCHAR(50),         -- Employee's last name
    manager_id INT                 -- Manager's ID (links to another employee_id in the same table)
);

-- Insert records into employee table
INSERT INTO employee (employee_id, first_name, last_name, manager_id) VALUES
(1, 'John', 'Doe', NULL),          -- John Doe has no manager (CEO)
(2, 'Jane', 'Smith', 1),           -- Jane Smith reports to John Doe
(3, 'Michael', 'Johnson', 1),      -- Michael Johnson reports to John Doe
(4, 'Emily', 'Davis', 2),          -- Emily Davis reports to Jane Smith
(5, 'Robert', 'Brown', 3);         -- Robert Brown reports to Michael Johnson
```

Find the Manager of Each Employee

A **self join** is when a table is joined with itself. In this case, we are joining the `employee` table with itself to match each employee with their manager.

SELECT e.first_name AS Employee, m.first_name AS Manager

FROM employee e

LEFT JOIN employee m ON e.manager_id = m.employee_id;

Explanation (Line by Line)

1. **`SELECT e.first_name AS Employee, m.first_name AS Manager`**:
   - We select two columns from the self-joined table:
     - `e.first_name`: Refers to the employee's first name. This is aliased as **Employee**.
     - `m.first_name`: Refers to the manager's first name. This is aliased as **Manager**.
   - The `e` and `m` aliases represent two different "instances" of the same `employee` table.
2. **`FROM employee e`**:
   - The first instance of the `employee` table is aliased as **e**.
   - This represents the employees whose managers we are trying to find.
3. **`LEFT JOIN employee m ON e.manager_id = m.employee_id`**:
   - The second instance of the `employee` table is aliased as **m**.
   - This instance represents the managers.
   - We use `LEFT JOIN` because we want to include all employees, even if they don't have a manager (like the CEO).
   - **`ON e.manager_id = m.employee_id`**:
     - This is the join condition.
     - For each row in the `employee` table (alias `e`), we look for a matching row in the same table (alias `m`) where `e.manager_id` matches `m.employee_id`.
     - Essentially, we are saying: "For every employee, find the row where the employee's manager ID matches the employee ID of another person."

Result (Background Mechanism)

Let's break down how SQL processes this **self join** row by row:

- **Step 1**: The `employee` table (`e`) row for **John Doe** (employee_id 1) is considered.
  - Since John has no manager (`manager_id` is `NULL`), there is no match found in the `m` table (other employees). Therefore, the result will be:
    - **Employee**: John
    - **Manager**: `NULL` (John has no manager).
- **Step 2**: The `employee` table (`e`) row for **Jane Smith** (employee_id 2) is considered.

    - Jane's `manager_id` is 1, so SQL finds the row in the `employee` table (`m`) where `employee_id` is 1, which is **John Doe**.
    - The result will be:
        - **Employee**: Jane
        - **Manager**: John
- **Step 3**: The `employee` table (`e`) row for **Michael Johnson** (employee_id 3) is considered.
    - Michael's `manager_id` is also 1, so SQL again finds the row in the `employee` table (`m`) where `employee_id` is 1, which is **John Doe**.
    - The result will be:
        - **Employee**: Michael
        - **Manager**: John
- **Step 4**: The `employee` table (`e`) row for **Emily Davis** (employee_id 4) is considered.
    - Emily's `manager_id` is 2, so SQL finds the row in the `employee` table (`m`) where `employee_id` is 2, which is **Jane Smith**.
    - The result will be:
        - **Employee**: Emily
        - **Manager**: Jane
- **Step 5**: The `employee` table (`e`) row for **Robert Brown** (employee_id 5) is considered.
    - Robert's `manager_id` is 3, so SQL finds the row in the `employee` table (`m`) where `employee_id` is 3, which is **Michael Johnson**.
    - The result will be:
        - **Employee**: Robert
        - **Manager**: Michael