SQL Joins, Normalization (Day 3)

⇒ Note

- To show who is PK and FK open DB diagram then right click on relation then properties.
- If you restored DB of someone Diagrams will not be appeared so right click on DB ⇒ properties ⇒ Files ⇒ write in Owner (sa).

Joins types

1. **Cross Join**:

This query returns the Cartesian product of employees and departments.

```sql
-- Cartesian product (Cross Join) SELECT e.EmployeeName, d.DepartmentName FROM Employee e CROSS JOIN Department d;
```

2. Inner Join (Equi-Join):

This query fetches employees who belong to a department, using the foreign key `DepartmentID`.

```sql
-- Inner Join (Employees with Departments) SELECT e.EmployeeName, d.DepartmentName FROM Employee e INNER JOIN Department d ON e.DepartmentID = d.DepartmentID;
```

3. Outer Joins:

a. Left Outer Join:

Returns all employees and their departments, even if the employee is not assigned to any department (departments can be `NULL`).

```sql
-- Left Outer Join (All Employees with or without Department) SELECT e.EmployeeName, d.DepartmentName FROM Employee e LEFT OUTER JOIN Department d ON e.DepartmentID = d.DepartmentID;
```

b. Right Outer Join:

Returns all departments and their employees, even if no employee is assigned to that department (employees can be `NULL`).

```sql
-- Right Outer Join (All Departments with or without Employees) SELECT e.EmployeeName, d.DepartmentName FROM Employee e RIGHT OUTER JOIN Department d ON e.DepartmentID = d.DepartmentID;
```

c. Full Outer Join:

Returns all employees and departments, even if no matching pairs exist.

```sql
-- Full Outer Join (All Employees and Departments, even if no match) SELECT e.EmployeeName, d.DepartmentName FROM Employee e FULL OUTER JOIN Department d ON e.DepartmentID = d.DepartmentID;
```

4. Self Join (Unary Relationship):

This query returns employees and their managers. It assumes that employees are linked to their manager via the `ManagerID` in the same table.

```
-- Self Join (Employee and their Manager) SELECT e1.EmployeeName AS Employee, e2.EmployeeName AS Manager FROM Employee e1 INNER JOIN Employee e2 ON e1.ManagerID = e2.EmployeeID;
```

Some Quaries

1. Select employees, their projects, and working hours:

This query retrieves employee names, project names, and the hours they worked on each project.

```
-- Select employee names, project names, and their working hours SELECT e.EmployeeName, p.ProjectName, w.Hours FROM Employee e INNER JOIN Works_for w ON e.EmployeeID = w.EmployeeID INNER JOIN Project p ON p.ProjectID = w.ProjectID;
```

2. Select employee names, project names, hours, and department names:

This query retrieves employees, their projects, hours worked, and the departments they belong to.

```
-- Select employee names, project names, hours, and department names SELECT e.EmployeeName, p.ProjectName, w.Hours, d.DepartmentName FROM Employee e INNER JOIN Works_for w ON e.EmployeeID = w.EmployeeID INNER JOIN Project p ON p.ProjectID = w.ProjectID INNER JOIN Department d ON d.DepartmentID = e.DepartmentID;
```

3. Update hours for male employees:

Increase the working hours by 10 for male employees.

```
-- Update working hours for male employees UPDATE Works_for SET Hours += 10 FROM Employee e INNER JOIN Works_for w ON e.EmployeeID = w.EmployeeID WHERE e.Gender = 'M';
```

String Operations:

4. Using `ISNULL` to replace NULL values:

If `EmployeeName` is `NULL`, replace it with 'hi' (this does not update the actual value in the table).

```
-- Replace NULL values in EmployeeName with 'hi' SELECT ISNULL(EmployeeName, 'NoName') AS EmployeeName FROM Employee;
```

5. Using `COALESCE` for multi-level replacements:

If `EmployeeName` is `NULL`, replace it with `LastName`. If `LastName` is also `NULL`, use 'Omar'.

```
-- Multi-level replacement using COALESCE SELECT COALESCE(EmployeeName, LastName, 'Omar') AS EmployeeName FROM Employee
```

6. Concatenating different data types:

To concatenate different data types (such as string and numeric), you must convert numeric types to string.

```
-- Concatenate EmployeeName and Salary after converting Salary to varchar SELECT COALESCE(EmployeeName, 'hi') + ' ' + CONVERT(VARCHAR(10), Salary) AS EmployeeInfo FROM Employee;
```

7. Handle NULL in concatenation:

If one of the concatenated values is `NULL`, the result would be `NULL`, so handle this using `ISNULL`.

```
-- Handle NULL in concatenation for Salary SELECT COALESCE(EmployeeName, 'hi') + ' ' + CONVERT(VARCHAR(10), ISNULL(Salary, 0)) AS EmployeeInfo FROM Employee;
```

8. Using `CONCAT` for type-safe concatenation:

`CONCAT` automatically converts all types to string, replacing `NULL` with an empty string.

```
-- Use CONCAT to safely concatenate EmployeeName and Salary SELECT CONCAT(EmployeeName, ' ', Salary) AS EmployeeInfo FROM Employee;
```

Pattern Matching with `LIKE`:

9. Pattern matching examples:

> (_) is one char and (%) is zero or more char

```
-- 1. Select all employees where EmployeeName contains 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '%r%'; -- 2. Select employees where EmployeeName ends with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '%r'; -- 3. Select employees where EmployeeName starts with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE 'r%'; -- 4. Select employees where the second character is 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '_r%'; -- 5. Select employees where EmployeeName starts with 'o' and ends with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE 'o%r'; -- 6. Select employees where EmployeeName starts with 'o' or 'a' and ends with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '[oa]%r'; -- 7. Select employees where EmployeeName starts with anything but 'a', 'b', or 'c' and ends with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '[^abc]%r'; -- 8. Select employees where EmployeeName starts with a character between 'a' and 'h' and ends with 'r' SELECT * FROM Employee WHERE EmployeeName LIKE '[a-h]%r'; -- 9. Select employees where EmployeeName ends with a percent sign (%) SELECT * FROM Employee WHERE EmployeeName LIKE '%[%]'; -- 10. Select employees where EmployeeName contains an underscore (_) SELECT * FROM Employee WHERE EmployeeName LIKE '%[_]%'; --11.If you're trying to find employee names with at least two underscores, you'd write: SELECT * FROM Employee WHERE EmployeeName LIKE '%[_]%[_]%'; --12. find employee names begin and end with underscores, you'd write: SELECT * FROM Employee WHERE EmployeeName LIKE '[_] % [_]';
```

Ordering Results:

10. Order by column position:

Order results by the second column.

```
-- Order by the second column SELECT * FROM Employee ORDER BY 2;
```

11. Order by multiple columns:

Order by `EmployeeName` in ascending order, and if names are equal, order by `LastName` in descending order.

```sql
-- Order by EmployeeName (ascending) and LastName (descending) SELECT * FROM
Employee ORDER BY EmployeeName ASC, LastName DESC;
```

Key Improvements:

1. **Consistent naming**: Use descriptive and consistent naming for columns and tables.
2. **Joins**: Utilize explicit `INNER JOIN` and `OUTER JOIN` syntax for clarity.
3. **String operations**: Use `ISNULL`, `COALESCE`, and `CONCAT` appropriately for handling `NULL` values and concatenation.
4. **Pattern matching**: Clearly demonstrate usage of pattern matching with `LIKE`.
5. **Order by**: Use meaningful ordering, including column positions and multi-column sorting.

Normalization

Normalization: Minimizing Redundancy and Inconsistencies

Normalization is a fundamental process in database design that helps reduce redundancy and avoid data anomalies such as insertion, deletion, and modification issues. It involves breaking larger tables into smaller, more manageable related tables and establishing relationships between them.

Key Points of Normalization:

1. **DB Design Approaches**:
   - Databases can be designed using **Entity-Relationship Diagrams (ERD)** or by applying **Normalization** rules.
   - **ERD** provides a visual representation of the entities and relationships, while **Normalization** is a systematic approach for improving data integrity.
2. **Breaking Tables**:
   - In **Normalization**, you take a large, unstructured table and break it into two or more smaller tables, establishing relationships between them using primary and foreign keys.

3. **Stages of Normalization**:
   - **Normalization** follows a series of steps, each aimed at eliminating different types of redundancy and dependency issues. If a table violates certain rules, it is said to be **denormalized**.
4. **Working on Existing Data**:
   - Normalization can be applied to **data already existing** in a database, Excel sheets, or other sources to restructure it for better integrity and efficiency.
5. **Reinitialize DB for Correctness**:
   - By applying normalization, you ensure that the database tables are correctly structured and maintain the integrity of the data.
6. **Mapping Verification**:
   - You can check the correctness of table relationships by examining whether they follow the **Normalization rules**. This helps ensure that the structure avoids redundancy and anomalies.
7. **Avoiding Anomalies**:
   - **Insertion Anomaly**: Adding new rows forces the duplication of data.
   - **Deletion Anomaly**: Deleting a row may cause a loss of valuable data in the future.
   - **Modification Anomaly**: Changing data in one row forces changes in other rows due to duplicated information.

Benefits of Normalization:

- **Redundancy Removed**: Instructor data is no longer repeated for every student enrolled in a course.
- **Anomalies Avoided**:
  - **Insertion Anomaly**: You can now add a new student without duplicating instructor information.
  - **Deletion Anomaly**: Deleting a student's enrollment record does not delete the instructor.
  - **Modification Anomaly**: If the instructor's email changes, you only need to update it in one place.

Functional Dependencies

Functional dependencies (FDs) describe the relationships between columns in a database table. If column **A** determines column **B**, this is represented as **A** → **B**, meaning for every unique value of **A**, there is only one corresponding value of **B**.

---

1. Full Functional Dependency (FFD)

A full functional dependency occurs when a non-key attribute is fully dependent on the entire **composite key**. If you remove any part of the composite key, the dependency no longer holds.

Example:

In a company database, suppose we have a table for **project assignments** where both **EmployeeID** and **ProjectID** together determine how many **HoursWorked** an employee spends on a project. This is a full dependency because both columns are needed to determine the number of hours worked.

```
-- Table: ProjectAssignment CREATE TABLE ProjectAssignment ( EmployeeID INT, ProjectID INT, HoursWorked INT, PRIMARY KEY (EmployeeID, ProjectID) -- Composite Primary Key ); -- Insert Example Data INSERT INTO ProjectAssignment (EmployeeID, ProjectID, HoursWorked) VALUES (1, 101, 30), (2, 101, 20), (1, 102, 25), (2, 102, 15); -- Full Dependency: EmployeeID, ProjectID --> HoursWorked
```

In this case, **EmployeeID** and **ProjectID** together determine the **HoursWorked**, so the functional dependency is:

`(EmployeeID, ProjectID) → HoursWorked`

---

2. Partial Functional Dependency (PFD)

A partial functional dependency occurs when a non-key attribute depends on **part** of a composite key. This typically happens when a column depends on one of the components of the composite primary key but not the entire key.

Example:

Let's say we have a table where **EmployeeID** and **DepartmentID** together form a composite key, and each **DepartmentID** determines the **Manager** of that department. This represents a **partial dependency**.

```
-- Table: EmployeeDepartment CREATE TABLE EmployeeDepartment ( EmployeeID IN
T, DepartmentID INT, Manager VARCHAR(100), PRIMARY KEY (EmployeeID, Departmen
tID) -- Composite Primary Key ); -- Insert Example Data INSERT INTO EmployeeD
epartment (EmployeeID, DepartmentID, Manager) VALUES (1, 1, 'John Smith'),
(2, 1, 'John Smith'), (3, 2, 'Mary Jones'); -- Partial Dependency: Department
ID --> Manager
```

Here, the manager is determined solely by **DepartmentID**, not by **EmployeeID**. Therefore, there is a **partial dependency**:

`DepartmentID → Manager`

To resolve this, we could move the **Manager** column to a separate **Department** table, making it fully dependent on **DepartmentID**:

```
-- Table: Department CREATE TABLE Department ( DepartmentID INT PRIMARY KEY,
Manager VARCHAR(100) ); -- Insert Data INSERT INTO Department (DepartmentID,
Manager) VALUES (1, 'John Smith'), (2, 'Mary Jones');
```

---

3. Transitive Functional Dependency (TFD)

A transitive dependency occurs when a non-key attribute depends on another non-key attribute, which is itself determined by the primary key. This creates an indirect relationship.

Example:

Suppose we have a table where **EmployeeID** determines the **ZipCode**, and the **ZipCode** determines the **City**. This is a **transitive dependency** because **City** depends on **ZipCode**, which is determined by **EmployeeID**.

```
-- Table: Employee CREATE TABLE Employee ( EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(100), ZipCode VARCHAR(10), City VARCHAR(100) ); -- Insert Example Data INSERT INTO Employee (EmployeeID, EmployeeName, ZipCode, City) VALUES (1, 'Alice', '90210', 'Beverly Hills'), (2, 'Bob', '10001', 'New York'), (3, 'Charlie', '94105', 'San Francisco'); -- Transitive Dependency: ZipCode --> City
```

In this case, **City** depends on **ZipCode**, and **ZipCode** is determined by **EmployeeID**, making this a transitive dependency:

`EmployeeID → ZipCode → City`

To remove the transitive dependency, we can create a separate **ZipCode** table that stores the **City**:

```
-- Table: ZipCode CREATE TABLE ZipCode ( ZipCode VARCHAR(10) PRIMARY KEY, City VARCHAR(100) ); -- Insert Data INSERT INTO ZipCode (ZipCode, City) VALUES ('90210', 'Beverly Hills'), ('10001', 'New York'), ('94105', 'San Francisco'); -- Modified Employee Table CREATE TABLE Employee ( EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(100), ZipCode VARCHAR(10), FOREIGN KEY (ZipCode) REFERENCES ZipCode(ZipCode) );
```

Now, **ZipCode** determines **City** in its own table, and **EmployeeID** only determines the **ZipCode**, avoiding the transitive dependency.

Summary of Functional Dependencies in SQL:

- **Full Functional Dependency**: Occurs when a non-key column depends on the entire composite primary key.
- **Partial Functional Dependency**: Occurs when a non-key column depends on part of the composite primary key.
- **Transitive Functional Dependency**: Occurs when a non-key column depends on another non-key column, which is itself determined by the primary key.

SID	SName	Birthdate	City	Zip Code	Subject	Grade	Teacher
1	Ahmed	1/1/1980	Cairo	1010	DB	A	Hany
1	Ahmed	1/1/1980	Cairo	1010	Math	B	Eman
1	Ahmed	1/1/1980	Cairo	1010	WinXP	A	khalid
2	Ali	1/1/1983	Alex	1111	DB	B	Hany
2	Ali	1/1/1983	Alex	1111	SWE	B	Heba
3	Mohamed	1/1/1990	Cairo	1010	NC	C	Mona

Full Functional Dependency — Sid,Subject → Grade

Partial Functional Dependency — Sid → SName
Subect →Teacher

Transitive Functional Dependency — ZipCode → City

Steps in Normalization

Step 1: Zero Normal Form (0NF)

At this stage, the table is unnormalized. It contains multivalued attributes or repeating groups.

Example: Employee Table in 0NF

Here, the employee table contains repeating groups (multiple phone numbers for the same employee) and is in **0NF**.

```
-- Table: Employee in 0NF CREATE TABLE Employee ( EmployeeID INT, EmployeeName VARCHAR(100), Department VARCHAR(100), PhoneNumber VARCHAR(100), -- Repeating group (multivalued attribute) Address VARCHAR(255) ); -- Insert Example Data INSERT INTO Employee (EmployeeID, EmployeeName, Department, PhoneNumber, Address) VALUES (1, 'Alice', 'HR', '12345, 67890', '123 Main St'), (2, 'Bob', 'Sales', '98765', '456 Oak St');
```

The **PhoneNumber** column contains multiple values, which violates normalization rules.

Step 2: First Normal Form (1NF)

In **1NF**, we remove repeating groups and multivalued attributes. Each column must contain atomic values (a single value per cell).

Transform to 1NF:

We split the multivalued attribute **PhoneNumber** into multiple rows.

```
-- Table: Employee in 1NF CREATE TABLE Employee ( EmployeeID INT, EmployeeName VARCHAR(100), Department VARCHAR(100), Address VARCHAR(255) ); CREATE TABLE EmployeePhone ( EmployeeID INT, PhoneNumber VARCHAR(100), PRIMARY KEY (EmployeeID, PhoneNumber), FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ); -- Insert Example Data into Employee INSERT INTO Employee (EmployeeID, EmployeeName, Department, Address) VALUES (1, 'Alice', 'HR', '123 Main St'), (2, 'Bob', 'Sales', '456 Oak St'); -- Insert Example Data into EmployeePhone INSERT INTO EmployeePhone (EmployeeID, PhoneNumber) VALUES (1, '12345'), (1, '67890'), (2, '98765');
```

Now, the **PhoneNumber** column contains only atomic values, and each phone number is linked to a single employee in a separate table.

Step 3: Second Normal Form (2NF)

In **2NF**, we remove **partial dependencies**. A table is in 2NF if it is in 1NF and all non-key attributes are fully dependent on the primary key. If we have a composite key, no non-key column should depend on part of that key.

Example:

Let's say an employee works on multiple projects, and the **ProjectID** and **EmployeeID** together form the primary key. We will move any attributes that depend on only part of the composite key to a new table.

Unnormalized Table (1NF):

```
-- Unnormalized EmployeeProject Table in 1NF CREATE TABLE EmployeeProject ( EmployeeID INT, ProjectID INT, ProjectName VARCHAR(100), HoursWorked INT, Department VARCHAR(100), -- Partial dependency: Department depends on EmployeeID, not ProjectID PRIMARY KEY (EmployeeID, ProjectID) );
```

Here, the **Department** depends only on **EmployeeID** (part of the composite key), which violates 2NF.

Transform to 2NF:

We move **Department** to a separate **Employee** table because it depends only on **EmployeeID**.

```
-- Table: Employee in 2NF CREATE TABLE Employee ( EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(100), Department VARCHAR(100), -- No partial dependency now Address VARCHAR(255) ); -- Table: EmployeeProject in 2NF CREATE TABLE EmployeeProject ( EmployeeID INT, ProjectID INT, ProjectName VARCHAR(100), HoursWorked INT, PRIMARY KEY (EmployeeID, ProjectID), FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ); -- Insert Example Data INSERT INTO Employee (EmployeeID, EmployeeName, Department, Address) VALUES (1, 'Alice', 'HR', '123 Main St'), (2, 'Bob', 'Sales', '456 Oak St'); INSERT INTO EmployeeProject (EmployeeID, ProjectID, ProjectName, HoursWorked) VALUES (1, 101, 'Project A', 30), (2, 102, 'Project B', 20);
```

Now, **Department** has been moved to the **Employee** table because it only depends on **EmployeeID**, achieving 2NF.

Step 4: Third Normal Form (3NF)

In **3NF**, we remove **transitive dependencies**. A table is in 3NF if it is in 2NF and there are no transitive dependencies (non-key columns depending on other non-key columns).

Example:

If **ZipCode** determines **City**, and **EmployeeID** determines **ZipCode**, then **City** depends on a non-key column (**ZipCode**), creating a transitive dependency.

Unnormalized Table (2NF):

```
-- Unnormalized Employee Table in 2NF CREATE TABLE Employee ( EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(100), Department VARCHAR(100), Address VARCHAR(255), ZipCode VARCHAR(10), -- Transitive dependency: City depends on ZipCode City VARCHAR(100) );
```

Here, **City** depends on **ZipCode**, which is not part of the primary key.

Transform to 3NF:

We move **City** to a separate table where it is determined by **ZipCode**, eliminating the transitive dependency.

```
-- Table: Employee in 3NF CREATE TABLE Employee ( EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(100), Department VARCHAR(100), Address VARCHAR(255), ZipCode VARCHAR(10), FOREIGN KEY (ZipCode) REFERENCES ZipCodeTable(ZipCode) ); -- Table: ZipCodeTable CREATE TABLE ZipCodeTable ( ZipCode VARCHAR(10) PRIMARY KEY, City VARCHAR(100) ); -- Insert Example Data into Employee INSERT INTO Employee (EmployeeID, EmployeeName, Department, Address, ZipCode) VALUES (1, 'Alice', 'HR', '123 Main St', '90210'), (2, 'Bob', 'Sales', '456 Oak St', '10001'); -- Insert Example Data into ZipCodeTable INSERT INTO ZipCodeTable (ZipCode, City) VALUES ('90210', 'Beverly Hills'), ('10001', 'New York');
```

Now, **City** depends on **ZipCode** in a separate table, and **Employee** only determines **ZipCode**, achieving 3NF.

Summary of Normalization Process:

1. **0NF**: Raw table with multivalued attributes and repeating groups.
2. **1NF**: Remove multivalued attributes, ensuring each cell has atomic values.
3. **2NF**: Eliminate partial dependencies by moving attributes to new tables if they depend only on part of a composite primary key.
4. **3NF**: Remove transitive dependencies by ensuring that non-key attributes depend only on the primary key.

Each step reduces redundancy and improves data integrity, ensuring a clean and optimized database design.

LAB

```
-- Create Database CREATE DATABASE CompanyDB; -- Use the CompanyDB USE
CompanyDB; -- Create Departments Table CREATE TABLE Departments (
DepartmentID INT PRIMARY KEY, -- Unique identifier for each department
DepartmentName VARCHAR(100) NOT NULL, -- Name of the department ManagerID
INT, -- ID of the department manager --FOREIGN KEY (ManagerID) REFERENCES
Employees(EmployeeID) -- Reference to Employees table ); -- Step 1: Alter the
Departments table to add the foreign key constraint ALTER TABLE Departments
ADD CONSTRAINT fk_Manager FOREIGN KEY (ManagerID) REFERENCES
Employees(EmployeeID); -- Create Employees Table CREATE TABLE Employees (
EmployeeID INT PRIMARY KEY , -- Unique identifier for each employee SSN
VARCHAR(11) NOT NULL UNIQUE, -- Social Security Number (unique) EmployeeName
VARCHAR(100) NOT NULL, -- Employee's name SupervisorSSN VARCHAR(11), --
Supervisor's SSN (can be NULL) Salary DECIMAL(10, 2) CHECK (Salary >= 0), --
Employee's salary Birthdate DATE, -- Employee's birth date Address
VARCHAR(255), -- Employee's address DepartmentID INT, -- ID of the department
employee belongs to FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID) -- Reference to Departments table ); -- Create
Projects Table CREATE TABLE Projects ( ProjectID INT PRIMARY KEY, -- Unique
identifier for each project ProjectName VARCHAR(100) NOT NULL, -- Name of the
project Location VARCHAR(100), -- Location of the project DepartmentID INT, -
- ID of the controlling department FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID) -- Reference to Departments table ); -- Create
Dependents Table CREATE TABLE Dependents ( DependentID INT PRIMARY KEY, --
Unique identifier for each dependent EmployeeID INT, -- ID of the employee
the dependent belongs to DependentName VARCHAR(100) NOT NULL, -- Name of the
dependent Relationship VARCHAR(50), -- Relationship to the employee Birthdate
DATE, -- Birthdate of the dependent FOREIGN KEY (EmployeeID) REFERENCES
Employees(EmployeeID) -- Reference to Employees table ); -- Create Works_For
Table (Associative Table for Employee-Project relationship) CREATE TABLE
Works_For ( EmployeeID INT, -- ID of the employee ProjectID INT, -- ID of the
project HoursPerWeek INT CHECK (HoursPerWeek >= 0), -- Hours per week
employee works on the project PRIMARY KEY (EmployeeID, ProjectID), --
Composite primary key FOREIGN KEY (EmployeeID) REFERENCES
Employees(EmployeeID), -- Reference to Employees table FOREIGN KEY
(ProjectID) REFERENCES Projects(ProjectID) -- Reference to Projects table );
INSERT INTO Departments (DepartmentID, DepartmentName, ManagerID) VALUES (1,
'Human Resources', NULL), (2, 'Finance', NULL), (3, 'IT', NULL), (4,
'Marketing', NULL), (5, 'Sales', NULL), (6, 'Customer Support', NULL), (7,
'Research and Development', NULL), (8, 'Production', NULL), (9, 'Logistics',
NULL), (10, 'Legal', NULL), (11, 'Purchasing', NULL), (12, 'Public
Relations', NULL), (13, 'Training', NULL), (14, 'Quality Assurance', NULL),
(15, 'Safety', NULL), (16, 'IT Support', NULL), (17, 'Administration', NULL),
```

```
(18, 'Business Development', NULL), (19, 'Data Analytics', NULL), (20,
'Product Management', NULL); INSERT INTO Employees (EmployeeID, SSN,
EmployeeName, SupervisorSSN, Salary, Birthdate, Address, DepartmentID) VALUES
(1, '123-45-6789', 'John Smith', NULL, 55000, '1985-06-15', '123 Main St,
Cityville', 1), (2, '234-56-7890', 'Jane Doe', '123-45-6789', 60000, '1983-
05-10', '456 Elm St, Cityville', 2), (3, '345-67-8901', 'Michael Johnson',
'123-45-6789', 50000, '1987-04-12', '789 Oak St, Cityville', 3), (4, '456-78-
9012', 'Emily Davis', '234-56-7890', 70000, '1990-03-08', '321 Pine St,
Cityville', 4), (5, '567-89-0123', 'Chris Wilson', '234-56-7890', 48000,
'1989-07-22', '654 Maple St, Cityville', 5), (6, '678-90-1234', 'Anna Brown',
'345-67-8901', 52000, '1991-08-30', '987 Cedar St, Cityville', 6), (7, '789-
01-2345', 'James Taylor', '456-78-9012', 58000, '1988-09-19', '654 Birch St,
Cityville', 7), (8, '890-12-3456', 'Linda Anderson', '567-89-0123', 59000,
'1986-11-15', '321 Walnut St, Cityville', 8), (9, '901-23-4567', 'David
Thomas', '678-90-1234', 64000, '1984-12-05', '123 Spruce St, Cityville', 9),
(10, '012-34-5678', 'Sarah Martinez', '789-01-2345', 62000, '1992-10-25',
'456 Ash St, Cityville', 10), (11, '123-45-6780', 'Daniel Garcia', '890-12-
3456', 53000, '1980-02-14', '789 Fir St, Cityville', 11), (12, '234-56-7891',
'Jessica Rodriguez', '901-23-4567', 47000, '1993-01-30', '321 Cherry St,
Cityville', 12), (13, '345-67-8902', 'Thomas Lee', '012-34-5678', 55000,
'1995-03-20', '654 Peach St, Cityville', 13), (14, '456-78-9013', 'Laura
Wilson', '123-45-6780', 61000, '1982-06-11', '987 Plum St, Cityville', 14),
(15, '567-89-0124', 'Brian Martin', '234-56-7891', 49000, '1988-07-29', '654
Willow St, Cityville', 15), (16, '678-90-1235', 'Megan Clark', '345-67-8902',
57000, '1984-08-20', '321 Poppy St, Cityville', 16), (17, '789-01-2346',
'Patricia Lewis', '456-78-9013', 56000, '1991-09-09', '123 Lotus St,
Cityville', 17), (18, '890-12-3457', 'Kevin Walker', '567-89-0124', 50000,
'1983-10-31', '456 Rose St, Cityville', 18), (19, '901-23-4568', 'Amy Hall',
'678-90-1235', 48000, '1990-11-20', '789 Daisy St, Cityville', 19), (20,
'012-34-5679', 'Mark Allen', '789-01-2346', 45000, '1986-12-12', '321 Lily
St, Cityville', 20); INSERT INTO Projects (ProjectID, ProjectName, Location,
DepartmentID) VALUES (1, 'Project Alpha', 'Cairo', 1), (2, 'Project Beta',
'Alexandria', 2), (3, 'Project Gamma', 'Cairo', 3), (4, 'Project Delta',
'Alexandria', 4), (5, 'Project Epsilon', 'Cairo', 5), (6, 'Project Zeta',
'Alexandria', 6), (7, 'Project Eta', 'Cairo', 7), (8, 'Project Theta',
'Alexandria', 8), (9, 'Project Iota', 'Cairo', 9), (10, 'Project Kappa',
'Alexandria', 10), (11, 'Project Lambda', 'Cairo', 11), (12, 'Project Mu',
'Alexandria', 12), (13, 'Project Nu', 'Cairo', 13), (14, 'Project Xi',
'Alexandria', 14), (15, 'Project Omicron', 'Cairo', 15), (16, 'Project Pi',
'Alexandria', 16), (17, 'Project Rho', 'Cairo', 17), (18, 'Project Sigma',
'Alexandria', 18), (19, 'Project Tau', 'Cairo', 19), (20, 'Project Upsilon',
'Alexandria', 20); INSERT INTO Dependents (DependentID, EmployeeID,
DependentName, Relationship, Birthdate) VALUES (1, 1, 'Alice Smith',
'Daughter', '2010-05-14'), (2, 1, 'Bob Smith', 'Son', '2012-11-23'), (3, 2,
```

```
'Charlie Doe', 'Son', '2011-07-19'), (4, 3, 'Daniel Johnson', 'Brother',
'1985-02-15'), (5, 4, 'Eva Davis', 'Sister', '1995-03-18'), (6, 5, 'Fiona
Wilson', 'Daughter', '2008-04-25'), (7, 6, 'George Brown', 'Son', '2015-01-
10'), (8, 7, 'Hannah Taylor', 'Daughter', '2013-06-12'), (9, 8, 'Ian
Anderson', 'Son', '2014-08-22'), (10, 9, 'Jack Thomas', 'Brother', '1980-09-
30'), (11, 10, 'Karen Martinez', 'Sister', '1992-10-14'), (12, 11, 'Larry
Garcia', 'Brother', '1988-01-05'), (13, 12, 'Megan Rodriguez', 'Daughter',
'2007-12-20'), (14, 13, 'Nancy Lee', 'Sister', '1991-11-25'), (15, 14, 'Oscar
Wilson', 'Son', '2015-03-17'), (16, 15, 'Paula Martin', 'Daughter', '2016-07-
28'), (17, 16, 'Quinn Clark', 'Daughter', '2012-04-05'), (18, 17, 'Ryan
Lewis', 'Son', '2014-06-30'), (19, 18, 'Sophie Hall', 'Daughter', '2011-09-
09'), (20, 19, 'Tom Allen', 'Son', '2009-12-12'); INSERT INTO Works_For
(EmployeeID, ProjectID, HoursPerWeek) VALUES (1, 1, 40), (1, 2, 20), (2, 1,
30), (2, 3, 25), (3, 4, 35), (3, 5, 20), (4, 6, 40), (5, 7, 15), (6, 8, 30),
(7, 9, 20), (8, 10, 35), (9, 11, 40), (10, 12, 30), (11, 13, 25), (12, 14,
20), (13, 15, 15), (14, 16, 40), (15, 17, 30), (16, 18, 35), (17, 19, 20),
(18, 20, 25);
```

Below are the SQL queries to execute the requested operations on the `CompanyDB` database.

Query 1: Display the Department ID, Name, and ID and Name of its Manager

```
SELECT d.DepartmentID, d.DepartmentName, e.EmployeeID AS ManagerID, e.Employe
eName AS ManagerName FROM Departments d LEFT JOIN Employees e ON d.ManagerID
= e.EmployeeID;
```

Query 2: Display the Name of the Departments and the Name of the Projects Under Its Control

```
SELECT d.DepartmentName, p.ProjectName FROM Departments d JOIN Projects p ON
d.DepartmentID = p.DepartmentID;
```

Query 3: Display Full Data About All the Dependents Associated with the Name of the Employee They Depend On

```
SELECT e.EmployeeName AS EmployeeName, d.* FROM Dependents d JOIN Employees e
ON d.EmployeeID = e.EmployeeID;
```

Query 4: Display the ID, Name, and Location of the Projects in Cairo or Alexandria

```
SELECT ProjectID, ProjectName, Location FROM Projects WHERE Location IN ('Cai
ro', 'Alexandria');
```

Query 5: Display the Full Data of the Projects with a Name Starting with "A"

```
SELECT * FROM Projects WHERE ProjectName LIKE 'A%';
```

Query 6: Display All Employees in Department 30 Whose Salary is Between 1000 and 2000 LE Monthly

```
SELECT * FROM Employees WHERE DepartmentID = 30 AND Salary BETWEEN 1000 AND 2
000;
```

Query 7: Retrieve Names of All Employees in Department 10 Who Work More Than or Equal to 10 Hours Per Week on "AL Rabwah" Project

```
SELECT e.EmployeeName FROM Employees e JOIN Works_For wf ON e.EmployeeID = wf.EmployeeID JOIN Projects p ON wf.ProjectID = p.ProjectID WHERE e.DepartmentID = 10 AND wf.HoursPerWeek >= 10 AND p.ProjectName = 'AL Rabwah';
```

Query 8: Find Names of Employees Who Are Directly Supervised by Kamel Mohamed

```
SELECT e.EmployeeName FROM Employees e WHERE e.SupervisorSSN = (SELECT SSN FROM Employees WHERE EmployeeName = 'Kamel Mohamed');
```

Query 9: Retrieve Names of All Employees and the Names of the Projects They Are Working On, Sorted by Project Name

```
SELECT e.EmployeeName, p.ProjectName FROM Employees e JOIN Works_For wf ON e.EmployeeID = wf.EmployeeID JOIN Projects p ON wf.ProjectID = p.ProjectID ORDER BY p.ProjectName;
```

Query 10: For Each Project Located in Cairo City, Find Project Number, Controlling Department Name, Department Manager Last Name, Address, and Birthdate

```
SELECT p.ProjectID, d.DepartmentName, e.EmployeeName AS ManagerLastName, e.Address, e.Birthdate FROM Projects p JOIN Departments d ON p.DepartmentID = d.DepartmentID JOIN Employees e ON d.ManagerID = e.EmployeeID WHERE p.Location = 'Cairo';
```

Query 11: Display All Data of the Managers

```
SELECT * FROM Employees WHERE EmployeeID IN (SELECT DISTINCT ManagerID FROM D
epartments WHERE ManagerID IS NOT NULL);
```

Query 12: Display All Employees Data and the Data of Their Dependents Even If They Have No Dependents

```
SELECT e.*, d.* FROM Employees e LEFT JOIN Dependents d ON e.EmployeeID = d.E
mployeeID;
```

Data Manipulating Language

1. Insert Your Personal Data to the Employee Table as a New Employee in Department Number 30

```
INSERT INTO Employees (EmployeeID, SSN, EmployeeName, SupervisorSSN, Salary,
Birthdate, Address, DepartmentID) VALUES (21, '102672', 'Your Name', NULL, 30
00, 'YYYY-MM-DD', 'Your Address', 30);
```

Replace `'Your Name'`, `'YYYY-MM-DD'`, and `'Your Address'` with your actual data.

2. Insert Another Employee with Your Friend's Personal Data as New Employee in Department Number 30

```
INSERT INTO Employees (EmployeeID, SSN, EmployeeName, SupervisorSSN, Salary,
Birthdate, Address, DepartmentID) VALUES (22, '102660', 'Friend\\'s Name', NU
LL, NULL, 'YYYY-MM-DD', 'Friend\\'s Address', 30);
```

Replace `'Friend's Name'`, `'YYYY-MM-DD'`, and `'Friend's Address'` with your friend's actual data.

3. Upgrade Your Salary by 20% of Its Last Value

```sql
UPDATE Employees SET Salary = Salary * 1.2 WHERE SSN = '102672'; -- Your SSN
```