SYMBIOSIS INSTITUTE OF TECHNOLOGY, NAGPUR

DATABASE MANAGEMENT SYSTEMS

Salary Management System

COMPUTER SCIENCE AND ENGINEERING

BATCH 2022-26

Course Name- DBMS

Course Code- T7907

SEMESTER- 5

SECTION- A

NAME: Mahati Murthy Akella

PRN: 22070521027

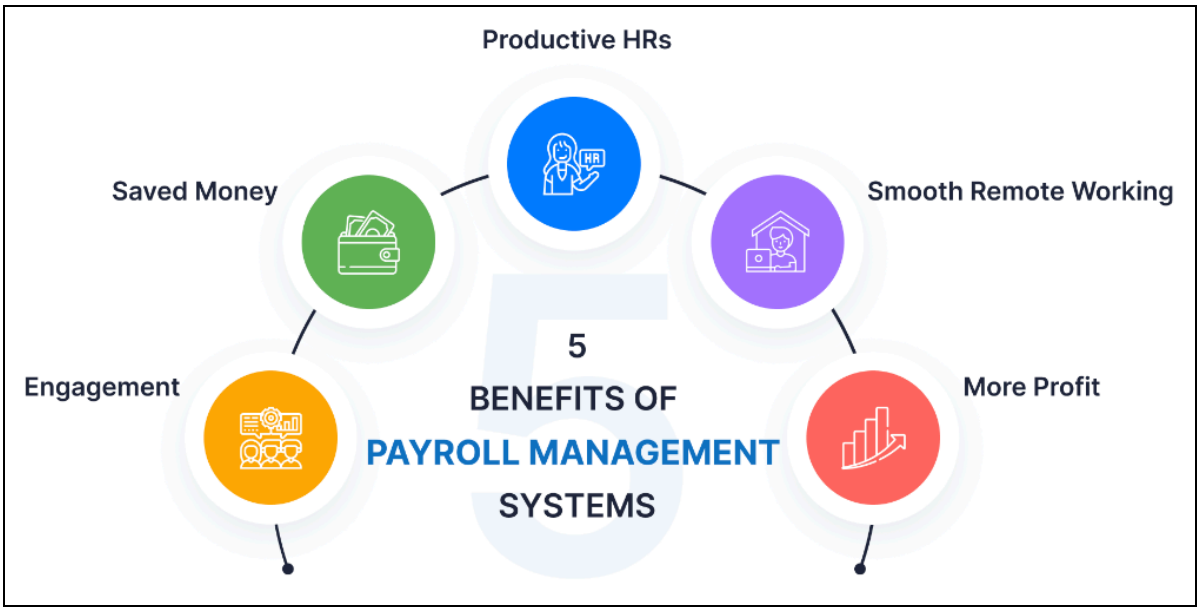
Project Report DBMS Mini Project

Table of Contents

I. Problem Statement

The **Salary Management System** is designed to simplify and automate payroll processes in an organization. It manages critical data such as employee records, salary details, bonuses, deductions, tax calculations, and payment transactions. One of the core functions of this system is to ensure that salaries are calculated accurately by considering various components such as basic pay, allowances, tax deductions, and bonuses. The system also tracks payments made to employees, ensuring that the correct amounts are paid on time and providing a historical record of all salary transactions.

This system significantly reduces the manual effort and errors associated with traditional payroll management. By automating tasks such as salary computation, tax withholding, and bonus distribution, it ensures that the payroll process is efficient and compliant with company policies and regulations. The ability to generate detailed payroll reports also helps human resource (HR) and finance teams in analyzing salary trends, tracking employee compensation, and ensuring that all payments are aligned with the organization's budgetary goals.



II. ER Diagram Explanation

The **Entity-Relationship (ER) diagram** is a vital tool for structuring a **Salary Management System** database, showing the connections between different entities and their relationships. It ensures an efficient database design by illustrating how various components like employees, salaries, leaves, and transactions are linked. For a **Salary Management System**, the ER diagram helps clarify the interactions between employees, their salary details, payment transactions, leaves, and external funds, establishing a clear framework for efficient payroll processing and data management.

The **relationships** depicted in the ER diagram demonstrate the flow of data between entities, such as the link between employees and the salaries they receive, employees and the leave they avail, or how salary payments are recorded in transactions. These relationships help ensure that all payroll-related interactions are effectively captured, allowing for seamless payroll management and financial transparency within the organization.

- **Entities: -**

1. **Employee**: Captures essential details about employees. Attributes include **E_ID** (Employee ID), **E_Name** (Employee Name), **E_email** (Employee Email), **Gender**, and **Join_Date**.
2. **Salary**: Represents the salary structure of employees with attributes like **S_ID** (Salary ID), **Basic** (Basic Pay), and **Allowance**.
3. **Transaction**: Tracks salary payments with attributes such as **T_ID** (Transaction ID), **Amount**, **T_Date** (Transaction Date), **S_month** (Salary Month), and **E_ID** (Employee ID).
4. **Leave**: Records leave information of employees, with attributes like **L_ID** (Leave ID), **E_ID** (Employee ID), and **Month**.
5. **Fund**: Captures data on funds, representing additional financial sources, with attributes like **F_ID** (Fund ID) and **Amount**.

- **Relationships:**

1. EMPLOYEE → EMPLOYEE_SALARY (One-to-Many)
2. EMPLOYEE → LEAVE (One-to-Many)
3. EMPLOYEE → TRANSACTION (One-to-Many)
4. EMPLOYEE_SALARY → SALARY (Many-to-One)
5. TRANSACTION → FUND (Many-to-One)

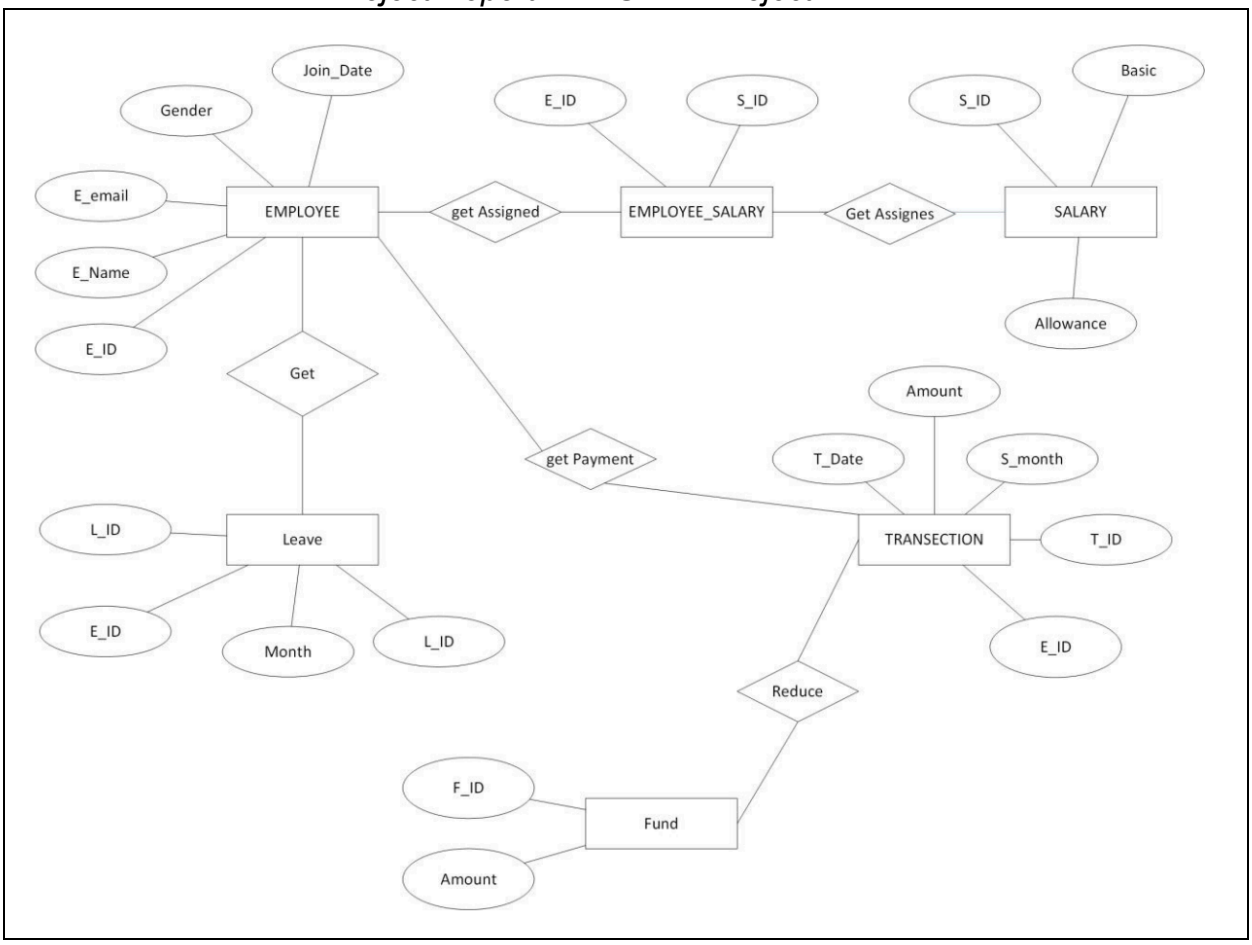


Fig.1 ER Diagram

III. Schema Diagram Explanation

The Schema Diagram is a crucial component of database design, providing a detailed blueprint of the database's structure. It outlines the organization of tables, columns, data types, and the relationships between these elements within the system. In contrast to the Entity-Relationship (ER) diagram we started with, which focused on conceptual relationships between entities, this schema diagram dives into the technical aspects, showing how the data is stored and managed in a database. It displays how tables are structured, how they relate through foreign keys, and how different constraints such as primary keys enforce data integrity.

In the context of this employee management and payroll system, the schema diagram defines how tables such as "EMPLOYEE," "SALARY," "LEAVE," "TRANSACTION," and "FUND" are designed, the columns they contain, and how these tables interconnect to ensure efficient data retrieval and management.

- **Tables: -**

1. **EMPLOYEE** table:

   Primary key: E_ID

   Other attributes: E_Name, E_email, Gender, Join_Date

2. **EMPLOYEE_SALARY** table:

   Composite primary key: E_ID, S_ID

   Foreign keys: E_ID references EMPLOYEE, S_ID references SALARY

3. **SALARY** table:

   Primary key: S_ID

   Other attributes: Basic, Allowance

4. **LEAVE** table:

   Primary key: L_ID

   Foreign key: E_ID references EMPLOYEE

   Other attribute: Month

5. **TRANSACTION** table:

   Primary key: T_ID

   Foreign key: E_ID references EMPLOYEE

   Other attributes: T_Date, Amount, S_month

6. **FUND** table:

   Primary key: F_ID

   Other attribute: Amount

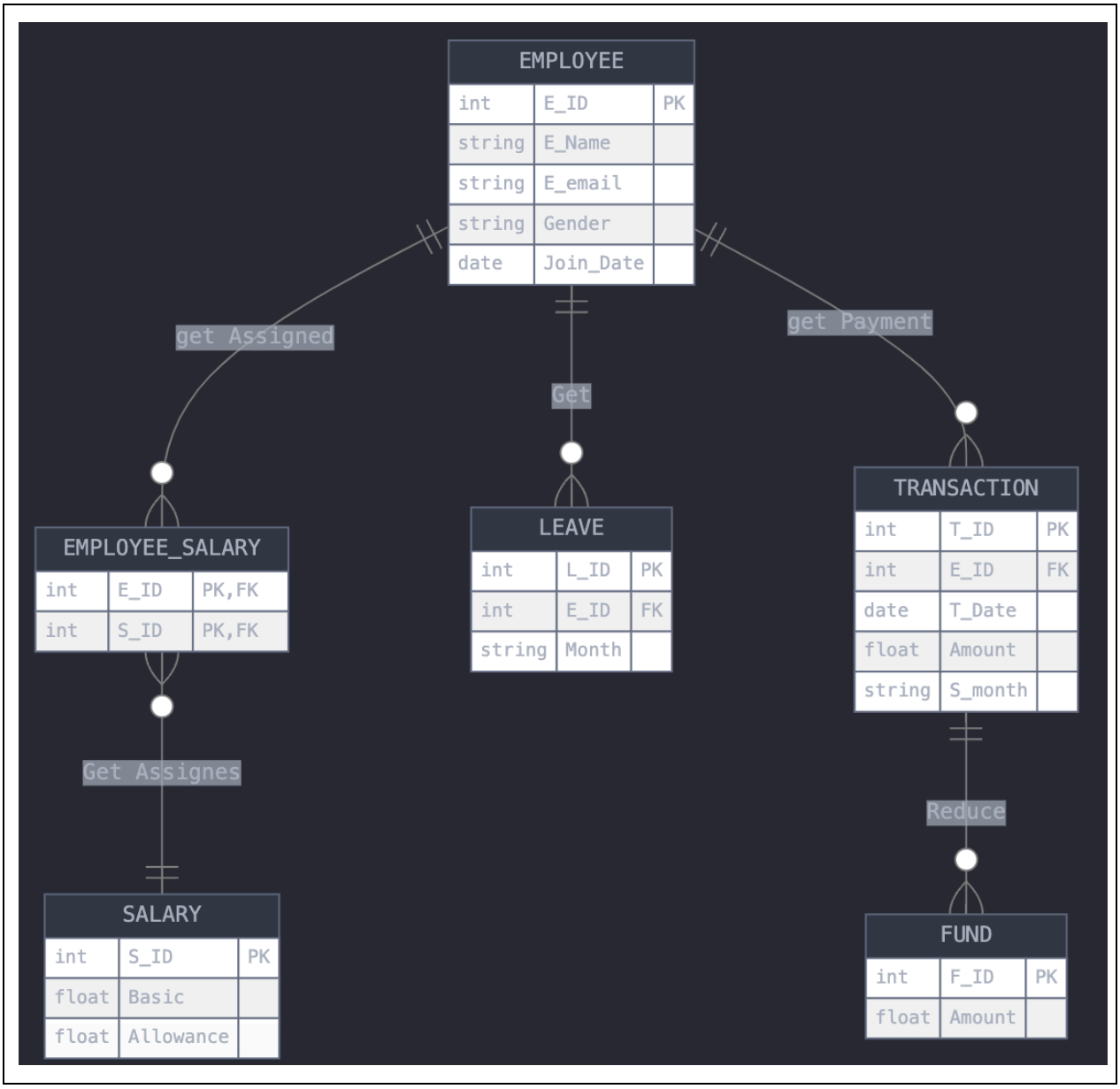


Fig.2 Schema Diagram

SQL Concepts Implemented in the Project

SQL (Structured Query Language) is the backbone of managing and manipulating data in a relational database system, making it essential for the operation of a comprehensive salary management system. In such a system, SQL is used to perform a wide range of tasks, including retrieving employee data, updating salary records, inserting new employee information, and archiving obsolete data. It enables users to interact with the database through queries to generate payroll reports, track employee performance, monitor leave balances, and manage detailed financial transactions.

The use of SQL in a salary management system allows for efficient handling of vast amounts of data, ensuring that essential business functions such as processing payroll, managing employee benefits, tracking leave, and maintaining accurate financial records are executed smoothly and accurately. This is crucial for organizations of all sizes to maintain compliance with labor laws and ensure employee satisfaction.

SQL concepts play a significant role in handling complex queries and ensuring data integrity within the salary management system:

1. **Joins:** Used to combine data from multiple tables, such as linking employee information with their corresponding salary structures and leave records.

2. Subqueries: Enable complex data retrieval, such as finding employees with the highest salaries or those who have taken the most leave in a given period.

3. **Triggers:** Automate processes like updating the TRANSACTION table when a new salary payment is made or adjusting the FUND table based on financial transactions.

4. Transactions: Ensure that critical operations, such as processing monthly payroll, are completed accurately and atomically, maintaining data consistency even in case of system failures.

5. **Views:** Create simplified or aggregated representations of data, useful for generating standardized reports on employee compensation or departmental salary expenditures.

6. **Stored Procedures:** Encapsulate complex business logic, such as calculating bonuses or processing year-end tax adjustments, ensuring consistent application across the system.

The project uses SQL queries to create, manipulate, and query tables as per the business requirements of the salary management system. All operations are performed using the MySQL terminal.

1. Table Employee Table : -

```
mysql> CREATE TABLE EMPLOYEE (
    ->     E_ID INT PRIMARY KEY,
    ->     E_Name VARCHAR(100) NOT NULL,
    ->     E_email VARCHAR(100) UNIQUE,
    ->     Gender ENUM('M', 'F', 'Other'),
    ->     Join_Date DATE
    -> );
Query OK, 0 rows affected (0.01 sec)
```

2. Table Salary: -

```
mysql> -- Create SALARY table
mysql> CREATE TABLE SALARY (
    ->     S_ID INT PRIMARY KEY,
    ->     Basic DECIMAL(10, 2) NOT NULL,
    ->     Allowance DECIMAL(10, 2) NOT NULL
    -> );
Query OK, 0 rows affected (0.05 sec)
```

3. Table Employee Salary: -

```
mysql> -- Create EMPLOYEE_SALARY junction table
mysql> CREATE TABLE EMPLOYEE_SALARY (
    ->     E_ID INT,
    ->     S_ID INT,
    ->     PRIMARY KEY (E_ID, S_ID),
    ->     FOREIGN KEY (E_ID) REFERENCES EMPLOYEE(E_ID),
    ->     FOREIGN KEY (S_ID) REFERENCES SALARY(S_ID)
    -> );
Query OK, 0 rows affected (0.01 sec)
```

4. Table Leave: -

```
mysql> -- Create LEAVE table
mysql> CREATE TABLE `LEAVE` (
    ->     L_ID INT PRIMARY KEY,
    ->     E_ID INT,
    ->     Month VARCHAR(10) NOT NULL,
    ->     FOREIGN KEY (E_ID) REFERENCES EMPLOYEE(E_ID)
    -> );
Query OK, 0 rows affected (0.01 sec)
```

5. Table Transaction and Fund: -

```
mysql> -- Create TRANSACTION table
mysql> CREATE TABLE `TRANSACTION` (
    ->     T_ID INT PRIMARY KEY,
    ->     E_ID INT,
    ->     T_Date DATE NOT NULL,
    ->     Amount DECIMAL(10, 2) NOT NULL,
    ->     S_month VARCHAR(10) NOT NULL,
    ->     FOREIGN KEY (E_ID) REFERENCES EMPLOYEE(E_ID)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> -- Create FUND table
mysql> CREATE TABLE FUND (
    ->     F_ID INT PRIMARY KEY,
    ->     Amount DECIMAL(10, 2) NOT NULL
    -> );
Query OK, 0 rows affected (0.00 sec)
```

Table Insertion: -

For **MySQL**, table insertion is performed using the `INSERT INTO` statement, which allows you to add new rows of data into a table by specifying values for each column. You can either insert data into all columns or specify particular columns for insertion.

```
mysql> INSERT INTO EMPLOYEE (E_ID, E_Name, E_email, Gender, Join_Date) VALUES
    -> (101, 'John Doe', 'john.doe@example.com', 'M', '2020-01-15'),
    -> (102, 'Jane Smith', 'jane.smith@example.com', 'F', '2019-03-10'),
    -> (103, 'Mike Brown', 'mike.brown@example.com', 'M', '2021-07-25'),
    -> (104, 'Alice Johnson', 'alice.johnson@example.com', 'F', '2020-11-05'),
    -> (105, 'Tom White', 'tom.white@example.com', 'M', '2022-04-14'),
    -> (106, 'Sara Parker', 'sara.parker@example.com', 'F', '2018-05-22'),
    -> (107, 'Chris Lee', 'chris.lee@example.com', 'Other', '2017-12-30');
Query OK, 7 rows affected (0.02 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> INSERT INTO SALARY (S_ID, Basic, Allowance) VALUES
    -> (1, 50000.00, 10000.00),
    -> (2, 60000.00, 12000.00),
    -> (3, 55000.00, 9000.00),
    -> (4, 70000.00, 15000.00),
    -> (5, 45000.00, 8000.00),
    -> (6, 80000.00, 20000.00),
    -> (7, 62000.00, 13000.00);
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> INSERT INTO EMPLOYEE_SALARY (E_ID, S_ID) VALUES
    -> (101, 1),
    -> (102, 2),
    -> (103, 3),
    -> (104, 4),
    -> (105, 5),
    -> (106, 6),
    -> (107, 7);
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

Project Report DBMS Mini Project

```
mysql> INSERT INTO `LEAVE` (L_ID, E_ID, Month) VALUES
    -> (1, 101, 'January'),
    -> (2, 102, 'February'),
    -> (3, 103, 'March'),
    -> (4, 104, 'April'),
    -> (5, 105, 'May'),
    -> (6, 106, 'June'),
    -> (7, 107, 'July');
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> INSERT INTO `TRANSACTION` (T_ID, E_ID, T_Date, Amount, S_month) VALUES
    -> (1, 101, '2024-01-31', 60000.00, 'January'),
    -> (2, 102, '2024-02-28', 72000.00, 'February'),
    -> (3, 103, '2024-03-31', 64000.00, 'March'),
    -> (4, 104, '2024-04-30', 85000.00, 'April'),
    -> (5, 105, '2024-05-31', 53000.00, 'May'),
    -> (6, 106, '2024-06-30', 100000.00, 'June'),
    -> (7, 107, '2024-07-31', 75000.00, 'July');
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> INSERT INTO FUND (F_ID, Amount) VALUES
    -> (1, 200000.00),
    -> (2, 250000.00),
    -> (3, 150000.00),
    -> (4, 300000.00),
    -> (5, 180000.00),
    -> (6, 500000.00),
    -> (7, 220000.00);
Query OK, 7 rows affected (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

Modification: -

We can use the `ALTER TABLE` statement to modify the structure of an existing table in the Salary Management System. For example:

```
mysql> ALTER TABLE SALARY
    -> MODIFY Bonus VARCHAR(20);
Query OK, 7 rows affected (0.03 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

This query adds a new column called Bonus to the Salaty.

- Updating Data in the Table: -

Once you've added the new **Bonus** column, you might want to modify some existing records to include values for the **Bonus** column in the **SALARY** table. Here's how you can update data in the **SALARY** table:

```
mysql> UPDATE SALARY
    -> SET Bonus = 'Year-End Bonus'
    -> WHERE S_ID = 2;  -- Update the record with S_ID = 2
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> UPDATE SALARY
    -> SET Bonus = 'Referral Bonus'
    -> WHERE S_ID = 3;  -- Update the record with S_ID = 3
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> UPDATE SALARY
    -> SET Bonus = 'Holiday Bonus'
    -> WHERE S_ID = 4;  -- Update the record with S_ID = 4
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> UPDATE SALARY
    -> SET Bonus = 'Signing Bonus'
    -> WHERE S_ID = 5;  -- Update the record with S_ID = 5
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> UPDATE SALARY
    -> SET Bonus = 'Retention Bonus'
    -> WHERE S_ID = 6;  -- Update the record with S_ID = 6
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

In these queries:

- The first `UPDATE` sets the **Bonus** to 'Performance Bonus' for the record where **S_ID = 1**.
- The second `UPDATE` sets the **Bonus** to 'Year-End Bonus' for the record where **S_ID = 2**.

```
mysql> UPDATE SALARY
    -> SET Bonus = 'Performance Bonus'
    -> WHERE S_ID = 1;  -- Update the record with S_ID = 1
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> UPDATE SALARY
    -> SET Bonus = 'Year-End Bonus'
    -> WHERE S_ID = 2;  -- Update the record with S_ID = 2
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

Deleting Rows and Dropping Tables: -

1. Create a Sample Table: -

Created a table Benefits that stores the benefits given top the employees

```
mysql> -- Create BENEFITS table
mysql> CREATE TABLE BENEFITS (
    ->     Benefit_ID INT PRIMARY KEY,
    ->     E_ID INT,
    ->     Benefit_Type VARCHAR(50) NOT NULL,
    ->     Benefit_Provider VARCHAR(100) NOT NULL,
    ->     FOREIGN KEY (E_ID) REFERENCES EMPLOYEE(E_ID)
    -> );
Query OK, 0 rows affected (0.04 sec)
```

This table has:

1. **Benefit_ID**: A unique identifier for each benefit entry, serving as the primary key.
2. **E_ID**: A foreign key that references the **E_ID** column from the **EMPLOYEE** table, establishing a relationship between the employee and their benefits.
3. **Benefit_Type**: A text field (up to 50 characters) that stores the type of benefit, such as health insurance or retirement plan.
4. **Benefit_Provider**: A text field (up to 100 characters) that stores the name of the provider offering the benefit.

2. Insert Rows into the Table:

Next, we will insert sample data into the Benefits table.

```
mysql> -- Insert 5 sample records into the BENEFITS table
mysql> INSERT INTO BENEFITS (Benefit_ID, E_ID, Benefit_Type, Benefit_Provider)
    -> VALUES
    -> (1, 101, 'Health Insurance', 'BlueCross HealthCare'),
    -> (2, 102, 'Retirement Plan', 'FutureSecure Pensions'),
    -> (3, 103, 'Health Insurance', 'Global Health Coverage'),
    -> (4, 104, 'Education Assistance', 'EduAssist Corp'),
    -> (5, 105, 'Life Insurance', 'LifeSecure Insurers');
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

This inserts 5 rows with benefit information for employees.

3. Delete a Row from the Table:

Let's delete the Benefits entry for Benefit_ID = 3.

```
mysql> DELETE FROM BENEFITS
    -> WHERE Benefit_ID = 3;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM BENEFITS;
+------------+------+---------------------+-----------------------+
| Benefit_ID | E_ID | Benefit_Type        | Benefit_Provider      |
+------------+------+---------------------+-----------------------+
|          1 |  101 | Health Insurance    | BlueCross HealthCare  |
|          2 |  102 | Retirement Plan     | FutureSecure Pensions |
|          4 |  104 | Education Assistance | EduAssist Corp       |
|          5 |  105 | Life Insurance      | LifeSecure Insurers   |
+------------+------+---------------------+-----------------------+
4 rows in set (0.00 sec)
```

This removes the Benefit record where Benefit_ID = 3.

4. Drop the Table:

Finally, let's drop the Benefit table.

```
mysql> DROP TABLE BENEFITS;
Query OK, 0 rows affected (0.02 sec)
```

Select Query Demonstrate: -

The SQL `SELECT` statement is used to query and retrieve data from a database. It allows you to select specific columns and rows of data from a table. In the context of a **Salary Management System**, the `SELECT *` statement can be used to retrieve all records and columns from the **EMPLOYEE** table.

```
mysql> SELECT * FROM EMPLOYEE;
+------+--------------+--------------------------+--------+------------+
| E_ID | E_Name       | E_email                  | Gender | Join_Date  |
+------+--------------+--------------------------+--------+------------+
|  101 | John Doe     | john.doe@example.com     | M      | 2020-01-15 |
|  102 | Jane Smith   | jane.smith@example.com   | F      | 2019-03-10 |
|  103 | Mike Brown   | mike.brown@example.com   | M      | 2021-07-25 |
|  104 | Alice Johnson| alice.johnson@example.com| F      | 2020-11-05 |
|  105 | Tom White    | tom.white@example.com    | M      | 2022-04-14 |
|  106 | Sara Parker  | sara.parker@example.com  | F      | 2018-05-22 |
|  107 | Chris Lee    | chris.lee@example.com    | Other  | 2017-12-30 |
+------+--------------+--------------------------+--------+------------+
7 rows in set (0.00 sec)
```

SQL Operators: -

The SQL operators include comparison operators, logical operators, and arithmetic operators.

Use of Comparison Operator: (=, <, >)

-- find all employees with a salary greater than a 30000 :

```
mysql> SELECT * FROM SALARY
    -> WHERE Basic > 30000;
+------+----------+-----------+-------------------+
| S_ID | Basic    | Allowance | Bonus             |
+------+----------+-----------+-------------------+
|    1 | 50000.00 |  10000.00 | Performance Bonus |
|    2 | 60000.00 |  12000.00 | Year-End Bonus    |
|    3 | 55000.00 |   9000.00 | Referral Bonus    |
|    4 | 70000.00 |  15000.00 | Holiday Bonus     |
|    5 | 45000.00 |   8000.00 | Signing Bonus     |
|    6 | 80000.00 |  20000.00 | Retention Bonus   |
|    7 | 62000.00 |  13000.00 | 0.00              |
+------+----------+-----------+-------------------+
7 rows in set (0.01 sec)
```

Use of Logical Operator: (AND, OR, NOT)

--Retrieving all records where the basic salary is greater than 30000 but is less than 50000 :

```
mysql> SELECT * FROM SALARY
    -> WHERE Basic > 30000 AND Basic < 50000;
+------+----------+-----------+---------------+
| S_ID | Basic    | Allowance | Bonus         |
+------+----------+-----------+---------------+
|    5 | 45000.00 |   8000.00 | Signing Bonus |
+------+----------+-----------+---------------+
1 row in set (0.00 sec)
```

Use of Arithmetic Operator: (+, -, *, /)

-- Calculate the difference between the highest **MAX** and lowest **MIN** basic salary in the **SALARY** table, aliasing it as `Salary_Difference`.

```
mysql> SELECT MAX(Basic) - MIN(Basic) AS Salary_Difference
    -> FROM SALARY;
+-------------------+
| Salary_Difference |
+-------------------+
|          35000.00 |
+-------------------+
1 row in set (0.01 sec)
```

Use of IN Operator

-- Find all orders placed by clients with IDs 1, 2, or 3

```
mysql> SELECT *
    -> FROM EMPLOYEE
    -> WHERE E_ID IN (1, 2, 3, 4, 5);
Empty set (0.00 sec)
```

Use of BETWEEN Operator

-- Find Employees with a Salary Within a Range:.

```
mysql> SELECT *
    -> FROM SALARY
    -> WHERE Basic BETWEEN 30000 AND 50000;
+------+----------+-----------+-------------------+
| S_ID | Basic    | Allowance | Bonus             |
+------+----------+-----------+-------------------+
|    1 | 50000.00 |  10000.00 | Performance Bonus |
|    5 | 45000.00 |   8000.00 | Signing Bonus     |
+------+----------+-----------+-------------------+
2 rows in set (0.00 sec)
```

Use of IS NULL Operator

-- Find Employees with Missing Email Addresses

```
mysql> SELECT *
    -> FROM EMPLOYEE
    -> WHERE E_email IS NULL;
Empty set (0.00 sec)
```

Use of NOT Operator

-- Find all cars that are not manufactured by 'Ford'.

```
mysql> SELECT *
    -> FROM SALARY
    -> WHERE Bonus IS NOT NULL;
+------+----------+-----------+-------------------+
| S_ID | Basic    | Allowance | Bonus             |
+------+----------+-----------+-------------------+
|    1 | 50000.00 |  10000.00 | Performance Bonus |
|    2 | 60000.00 |  12000.00 | Year-End Bonus    |
|    3 | 55000.00 |   9000.00 | Referral Bonus    |
|    4 | 70000.00 |  15000.00 | Holiday Bonus     |
|    5 | 45000.00 |   8000.00 | Signing Bonus     |
|    6 | 80000.00 |  20000.00 | Retention Bonus   |
|    7 | 62000.00 |  13000.00 | 0.00              |
+------+----------+-----------+-------------------+
7 rows in set (0.00 sec)
```

Explanation of Operators :

1. Comparison Operators:
   - =: Equal to.
   - >: Greater than.
   - <: Less than.

2. Logical Operators:
   - AND: Combines conditions where both must be true.
   - OR: Combines conditions where at least one must be true.
   - NOT: Negates a condition.

3. Arithmetic Operators:
   - Subtraction -
   - Addition +
   - Multiplication *

4. LIKE Operator:
   - Used for pattern matching.

5. IN Operator:
   - Matches any of a list of values.

6. BETWEEN Operator:
   - Used to filter data within a range of values.

7. IS NULL Operator:
   - Used to find null values.

8. NOT Operator:
   - Negates a condition.

SQL Joins: -

1. Natural Join

A NATURAL JOIN automatically joins tables based on columns with the same name and data type.

To combine the **EMPLOYEE** and **EMPLOYEE_SALARY** tables based on the `E_ID` column (which is common in both tables) :

```
mysql> SELECT *
    -> FROM EMPLOYEE
    -> NATURAL JOIN EMPLOYEE_SALARY;
+------+---------------+--------------------------+--------+------------+------+
| E_ID | E_Name        | E_email                  | Gender | Join_Date  | S_ID |
+------+---------------+--------------------------+--------+------------+------+
|  101 | John Doe      | john.doe@example.com     | M      | 2020-01-15 |    1 |
|  102 | Jane Smith    | jane.smith@example.com   | F      | 2019-03-10 |    2 |
|  103 | Mike Brown    | mike.brown@example.com   | M      | 2021-07-25 |    3 |
|  104 | Alice Johnson | alice.johnson@example.com| F      | 2020-11-05 |    4 |
|  105 | Tom White     | tom.white@example.com    | M      | 2022-04-14 |    5 |
|  106 | Sara Parker   | sara.parker@example.com  | F      | 2018-05-22 |    6 |
|  107 | Chris Lee     | chris.lee@example.com    | Other  | 2017-12-30 |    7 |
+------+---------------+--------------------------+--------+------------+------+
7 rows in set (0.01 sec)
```

2. Self Join

A SELF JOIN is when a table is joined with itself. This is useful when you need to compare rows within the same table.

3. Cross Join

A CROSS JOIN returns the Cartesian product of the two tables (i.e., every possible combination of rows).

`CROSS JOIN` between the `EMPLOYEE` and `SALARY` tables:

```
mysql> SELECT E.E_Name, S.Basic, S.Allowance
    -> FROM EMPLOYEE E
    -> CROSS JOIN SALARY S;
+---------------+----------+-----------+
| E_Name        | Basic    | Allowance |
+---------------+----------+-----------+
| Chris Lee     | 50000.00 |  10000.00 |
| Sara Parker   | 50000.00 |  10000.00 |
| Tom White     | 50000.00 |  10000.00 |
| Alice Johnson | 50000.00 |  10000.00 |
| Mike Brown    | 50000.00 |  10000.00 |
| Jane Smith    | 50000.00 |  10000.00 |
| John Doe      | 50000.00 |  10000.00 |
| Chris Lee     | 60000.00 |  12000.00 |
| Sara Parker   | 60000.00 |  12000.00 |
| Tom White     | 60000.00 |  12000.00 |
| Alice Johnson | 60000.00 |  12000.00 |
| Mike Brown    | 60000.00 |  12000.00 |
| Jane Smith    | 60000.00 |  12000.00 |
| John Doe      | 60000.00 |  12000.00 |
| Chris Lee     | 55000.00 |   9000.00 |
| Sara Parker   | 55000.00 |   9000.00 |
| Tom White     | 55000.00 |   9000.00 |
| Alice Johnson | 55000.00 |   9000.00 |
| Mike Brown    | 55000.00 |   9000.00 |
| Jane Smith    | 55000.00 |   9000.00 |
| John Doe      | 55000.00 |   9000.00 |
| Chris Lee     | 70000.00 |  15000.00 |
| Sara Parker   | 70000.00 |  15000.00 |
| Tom White     | 70000.00 |  15000.00 |
| Alice Johnson | 70000.00 |  15000.00 |
| Mike Brown    | 70000.00 |  15000.00 |
| Jane Smith    | 70000.00 |  15000.00 |
| John Doe      | 70000.00 |  15000.00 |
| Chris Lee     | 45000.00 |   8000.00 |
| Sara Parker   | 45000.00 |   8000.00 |
| Tom White     | 45000.00 |   8000.00 |
| Alice Johnson | 45000.00 |   8000.00 |
| Mike Brown    | 45000.00 |   8000.00 |
| Jane Smith    | 45000.00 |   8000.00 |
| John Doe      | 45000.00 |   8000.00 |
| Chris Lee     | 80000.00 |  20000.00 |
| Sara Parker   | 80000.00 |  20000.00 |
| Tom White     | 80000.00 |  20000.00 |
| Alice Johnson | 80000.00 |  20000.00 |
| Mike Brown    | 80000.00 |  20000.00 |
| Jane Smith    | 80000.00 |  20000.00 |
| John Doe      | 80000.00 |  20000.00 |
| Chris Lee     | 62000.00 |  13000.00 |
| Sara Parker   | 62000.00 |  13000.00 |
| Tom White     | 62000.00 |  13000.00 |
| Alice Johnson | 62000.00 |  13000.00 |
| Mike Brown    | 62000.00 |  13000.00 |
| Jane Smith    | 62000.00 |  13000.00 |
| John Doe      | 62000.00 |  13000.00 |
+---------------+----------+-----------+
49 rows in set (0.01 sec)
```

4. Left Outer Join

A LEFT OUTER JOIN returns all rows from the left table, and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.

```
mysql> SELECT E.E_Name, S.S_ID, S.Basic, S.Allowance
    -> FROM EMPLOYEE E
    -> LEFT OUTER JOIN EMPLOYEE_SALARY ES ON E.E_ID = ES.E_ID
    -> LEFT OUTER JOIN SALARY S ON ES.S_ID = S.S_ID;
+---------------+------+----------+-----------+
| E_Name        | S_ID | Basic    | Allowance |
+---------------+------+----------+-----------+
| John Doe      |    1 | 50000.00 |  10000.00 |
| Jane Smith    |    2 | 60000.00 |  12000.00 |
| Mike Brown    |    3 | 55000.00 |   9000.00 |
| Alice Johnson |    4 | 70000.00 |  15000.00 |
| Tom White     |    5 | 45000.00 |   8000.00 |
| Sara Parker   |    6 | 80000.00 |  20000.00 |
| Chris Lee     |    7 | 62000.00 |  13000.00 |
+---------------+------+----------+-----------+
7 rows in set (0.01 sec)
```

5. Right Outer Join

A RIGHT OUTER JOIN returns all rows from the right table, and the matching rows from the left table. If there is no match, NULL values are returned for columns from the left table.

```
mysql> SELECT E.E_Name, S.S_ID, S.Basic, S.Allowance
    -> FROM EMPLOYEE_SALARY ES
    -> RIGHT OUTER JOIN SALARY S ON ES.S_ID = S.S_ID
    -> LEFT OUTER JOIN EMPLOYEE E ON ES.E_ID = E.E_ID;
+---------------+------+----------+-----------+
| E_Name        | S_ID | Basic    | Allowance |
+---------------+------+----------+-----------+
| John Doe      |    1 | 50000.00 |  10000.00 |
| Jane Smith    |    2 | 60000.00 |  12000.00 |
| Mike Brown    |    3 | 55000.00 |   9000.00 |
| Alice Johnson |    4 | 70000.00 |  15000.00 |
| Tom White     |    5 | 45000.00 |   8000.00 |
| Sara Parker   |    6 | 80000.00 |  20000.00 |
| Chris Lee     |    7 | 62000.00 |  13000.00 |
+---------------+------+----------+-----------+
7 rows in set (0.00 sec)
```

6. Full Outer Join

A FULL OUTER JOIN returns all rows when there is a match in either table. Rows that do not have a match in one table will still appear in the result with NULL for the missing columns.

```
mysql> SELECT E.E_Name, S.S_ID, S.Basic, S.Allowance
    -> FROM EMPLOYEE E
    -> LEFT OUTER JOIN EMPLOYEE_SALARY ES ON E.E_ID = ES.E_ID
    -> LEFT OUTER JOIN SALARY S ON ES.S_ID = S.S_ID
    ->
    -> UNION
    ->
    -> SELECT E.E_Name, S.S_ID, S.Basic, S.Allowance
    -> FROM EMPLOYEE_SALARY ES
    -> RIGHT OUTER JOIN SALARY S ON ES.S_ID = S.S_ID
    -> LEFT OUTER JOIN EMPLOYEE E ON ES.E_ID = E.E_ID;
+--------------+------+----------+-----------+
| E_Name       | S_ID | Basic    | Allowance |
+--------------+------+----------+-----------+
| John Doe     |    1 | 50000.00 |  10000.00 |
| Jane Smith   |    2 | 60000.00 |  12000.00 |
| Mike Brown   |    3 | 55000.00 |   9000.00 |
| Alice Johnson|    4 | 70000.00 |  15000.00 |
| Tom White    |    5 | 45000.00 |   8000.00 |
| Sara Parker  |    6 | 80000.00 |  20000.00 |
| Chris Lee    |    7 | 62000.00 |  13000.00 |
+--------------+------+----------+-----------+
7 rows in set (0.01 sec)
```

First Part (Left Outer Join):

- This part of the query retrieves all records from the `EMPLOYEE` table along with any matching records from the `SALARY` table. If there are no matches, the `SALARY` fields will be `NULL`.

Second Part (Right Outer Join):

- This part retrieves all records from the `SALARY` table along with any matching records from the `EMPLOYEE` table. If there are no matches, the `EMPLOYEE` fields will be `NULL`.

UNION:

- The `UNION` operator combines the results of both queries, ensuring that you get a complete set of records from both tables.

7. Inner Join

An INNER JOIN returns only the rows where there is a match between both tables.

```
mysql> SELECT E.E_Name, S.Basic, S.Allowance
    -> FROM EMPLOYEE E
    -> INNER JOIN EMPLOYEE_SALARY ES ON E.E_ID = ES.E_ID
    -> INNER JOIN SALARY S ON ES.S_ID = S.S_ID;
+---------------+----------+-----------+
| E_Name        | Basic    | Allowance |
+---------------+----------+-----------+
| John Doe      | 50000.00 |  10000.00 |
| Jane Smith    | 60000.00 |  12000.00 |
| Mike Brown    | 55000.00 |   9000.00 |
| Alice Johnson | 70000.00 |  15000.00 |
| Tom White     | 45000.00 |   8000.00 |
| Sara Parker   | 80000.00 |  20000.00 |
| Chris Lee     | 62000.00 |  13000.00 |
+---------------+----------+-----------+
7 rows in set (0.00 sec)
```

Summary of Joins:

- **Natural Join**: Automatically matches columns with the same name.
- **Self Join**: Joins a table with itself.
- **Cross Join**: Returns the Cartesian product.
- **Left Outer Join**: Returns all rows from the left table, with NULL for missing matches.
- **Right Outer Join**: Returns all rows from the right table, with NULL for missing matches.
- **Full Outer Join**: Returns all rows when there is a match in either table.
- **Inner Join**: Returns only rows that have matches in both tables.

Functions: -

1. Function to Calculate Total Salary for an Employee

```
CREATE FUNCTION GetTotalSalary(EmpID INT)
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    DECLARE TotalSalary DECIMAL(10, 2);
    SELECT SUM(Basic + Allowance) INTO TotalSalary
    FROM SALARY
    JOIN EMPLOYEE_SALARY ON SALARY.S_ID = EMPLOYEE_SALARY.S_ID
    WHERE EMPLOYEE_SALARY.E_ID = EmpID;
    RETURN TotalSalary;
END;
```

2. Function to Get Leave Count for an Employee

```
CREATE FUNCTION GetLeaveCount(EmpID INT, LeaveMonth VARCHAR(10))
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE LeaveCount INT;
    SELECT COUNT(*) INTO LeaveCount
    FROM LEAVE
    WHERE E_ID = EmpID AND Month = LeaveMonth;
    RETURN LeaveCount;
END;
```

- The `DETERMINISTIC` keyword indicates that the function will return the same result any time it is called with the same input parameters.

3. Function to Record a New Transaction

```
CREATE FUNCTION RecordTransaction(EmpID INT, TransactionDate DATE, TransactionAmou
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
    INSERT INTO TRANSACTION (E_ID, T_Date, Amount, S_month)
    VALUES (EmpID, TransactionDate, TransactionAmount, TransactionMonth);
    RETURN 'Transaction recorded successfully';
END;
```

4. Function to Add a New Fund

```
CREATE FUNCTION AddNewFund(FundAmount DECIMAL(10, 2))
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
    INSERT INTO FUND (Amount)
    VALUES (FundAmount);
    RETURN 'Fund added successfully';
END;
```

Explanation:

1. **Calculate Total Salary for an Employee**: This function calculates the total salary for a given employee based on their basic salary and allowances.
2. **Get Leave Count for an Employee**: This function retrieves the number of leave records for a given employee in a specified month.
3. **Record a New Transaction**: This function inserts a new transaction record for an employee.
4. **Add a New Fund**: This function adds a new fund record to the FUND table.

Procedure: -

1. the `AddEmployee` procedure:

- Create the procedure:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE AddEmployee(
    ->     IN EmpID INT,
    ->     IN EmpName VARCHAR(100),
    ->     IN EmpEmail VARCHAR(100),
    ->     IN EmpGender ENUM('M', 'F', 'Other'),
    ->     IN JoinDate DATE
    -> )
    -> BEGIN
    ->     INSERT INTO EMPLOYEE (E_ID, E_Name, E_email, Gender, Join_Date)
    ->     VALUES (EmpID, EmpName, EmpEmail, EmpGender, JoinDate);
    -> END //
mysql> DELIMITER ;
Query OK, 0 rows affected (0.01 sec)
```

- Call the procedure to add an employee:

.

```
mysql> CALL AddEmployee(1, 'John Doe', 'john.doe@example.com', 'M', '2024-10-15');
Query OK, 1 row affected (0.01 sec)
```

- To verify the insertion run a SELECT query:

```
mysql> SELECT * FROM EMPLOYEE WHERE E_ID = 1;
+------+----------+---------------------+--------+------------+
| E_ID | E_Name   | E_email             | Gender | Join_Date  |
+------+----------+---------------------+--------+------------+
|    1 | John Doe | john.doe@example.com | M      | 2024-10-15 |
+------+----------+---------------------+--------+------------+
1 row in set (0.00 sec)
```

- If you try to insert a duplicate E_ID, you'll get an error:

```
mysql> CALL AddEmployee(1, 'Jane Doe', 'jane.doe@example.com', 'F', '2024-10-16');
ERROR 1062 (23000): Duplicate entry '1' for key 'EMPLOYEE.PRIMARY'
```

Explanation:

Procedure Creation:

- Defines `AddEmployee` procedure with 5 parameters
- Uses `INSERT INTO` to add new employee to `EMPLOYEE` table
- Output: `Query OK, 0 rows affected (0.01 sec)`
- Meaning: Procedure created successfully

Procedure Execution:

- Calls `AddEmployee` with sample data
- Output: `Query OK, 1 row affected (0.01 sec)`
- Meaning: One employee record inserted successfully

Verification:

- `SELECT` query retrieves inserted employee data
- Output: Shows row with inserted employee details
- Confirms successful insertion with correct data

Duplicate Insertion Attempt:

- Tries to insert employee with existing E_ID (1)
- Output: `ERROR 1062 (23000): Duplicate entry '1' for key 'EMPLOYEE.PRIMARY'`
- Demonstrates primary key constraint working correctly

Triggers: -

- This trigger will automatically insert a record into the EMPLOYEE_SALARY table whenever a new employee is added to the EMPLOYEE table.

```
CREATE TRIGGER after_employee_insert
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    INSERT INTO SALARY (S_ID, Basic, Allowance)
    VALUES (NEW.E_ID, 5000.00, 1000.00);

    INSERT INTO EMPLOYEE_SALARY (E_ID, S_ID)
    VALUES (NEW.E_ID, NEW.E_ID);
END;
```

```
mysql> CREATE TRIGGER after_employee_insert AFTER INSERT ON EMPLOYEE FOR EACH ROW
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO EMPLOYEE (E_ID, E_Name, E_email, Gender, Join_Date) VALUES (1,
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM EMPLOYEE WHERE E_ID = 1;
+------+----------+----------------------+--------+------------+
| E_ID | E_Name   | E_email              | Gender | Join_Date  |
+------+----------+----------------------+--------+------------+
|    1 | John Doe | john.doe@example.com | M      | 2024-10-15 |
+------+----------+----------------------+--------+------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM SALARY WHERE S_ID = 1;
+------+---------+-----------+
| S_ID | Basic   | Allowance |
+------+---------+-----------+
|    1 | 5000.00 |   1000.00 |
+------+---------+-----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM EMPLOYEE_SALARY WHERE E_ID = 1;
+------+------+
| E_ID | S_ID |
+------+------+
|    1 |    1 |
+------+------+
1 row in set (0.00 sec)
```

Explanation of the output:

1. The first attempt to create the trigger fails due to syntax error (missing delimiter).
2. The second attempt succeeds when the trigger is written in a single line.
3. An employee is inserted into the EMPLOYEE table.
4. The SELECT statements show that:
    - The employee was added to the EMPLOYEE table.
    - A corresponding salary record was automatically added to the SALARY table.
    - A linking record was automatically added to the EMPLOYEE_SALARY table.

- Trigger that updates the TRANSACTION table whenever a new salary record is inserted into the SALARY table. This trigger will record the initial salary as a transaction.

```
mysql> CREATE TRIGGER after_salary_insert AFTER INSERT ON SALARY FOR EACH ROW
BEGIN INSERT INTO TRANSACTION (T_ID, E_ID, T_Date, Amount, S_month) VALUES
(NEW.S_ID, NEW.S_ID, CURDATE(), NEW.Basic + NEW.Allowance, DATE_FORMAT(CURDATE(),
'%Y-%m')); END;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO EMPLOYEE (E_ID, E_Name, E_email, Gender, Join_Date) VALUES (2,
'Jane Doe', 'jane.doe@example.com', 'F', '2024-10-16');
Query OK, 1 row affected (0.01 sec)
```

```
mysql> SELECT * FROM EMPLOYEE WHERE E_ID = 2;
+------+----------+---------------------+--------+------------+
| E_ID | E_Name   | E_email             | Gender | Join_Date  |
+------+----------+---------------------+--------+------------+
|    2 | Jane Doe | jane.doe@example.com | F      | 2024-10-16 |
+------+----------+---------------------+--------+------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM SALARY WHERE S_ID = 2;
+------+---------+-----------+
| S_ID | Basic   | Allowance |
+------+---------+-----------+
|    2 | 5000.00 |   1000.00 |
+------+---------+-----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM EMPLOYEE_SALARY WHERE E_ID = 2;
+------+------+
| E_ID | S_ID |
+------+------+
|    2 |    2 |
+------+------+
1 row in set (0.00 sec)

mysql> SELECT * FROM TRANSACTION WHERE E_ID = 2;
+------+------+------------+---------+----------+
| T_ID | E_ID | T_Date     | Amount  | S_month  |
+------+------+------------+---------+----------+
|    2 |    2 | 2024-10-15 | 6000.00 | 2024-10  |
+------+------+------------+---------+----------+
1 row in set (0.00 sec)
```

Explanation of the output:

1. The trigger `after_salary_insert` is created successfully.
2. A new employee (Jane Doe) is inserted into the EMPLOYEE table.
3. The `after_employee_insert` trigger (created earlier) automatically adds records to the SALARY and EMPLOYEE_SALARY tables.
4. The new `after_salary_insert` trigger then automatically adds a record to the TRANSACTION table.
5. The SELECT statements show that:
   - The employee was added to the EMPLOYEE table.
   - A corresponding salary record was added to the SALARY table.
   - A linking record was added to the EMPLOYEE_SALARY table.
   - A transaction record was added to the TRANSACTION table, with the total salary amount (Basic + Allowance) and the current month.

View: -

1. View to Display Employee's Salary Details

```
CREATE VIEW EmployeeSalaryDetails AS
SELECT
    e.E_ID,
    e.E_Name,
    e.E_email,
    e.Gender,
    s.Basic,
    s.Allowance,
    (s.Basic + s.Allowance) AS Total_Salary
FROM
    EMPLOYEE e
JOIN
    EMPLOYEE_SALARY es ON e.E_ID = es.E_ID
JOIN
    SALARY s ON es.S_ID = s.S_ID;
```

Output :-

```
mysql> SELECT * FROM EmployeeSalaryDetails;
+-------+------------+----------------------+--------+--------+-----------+--------------+
| E_ID  | E_Name     | E_email              | Gender | Basic  | Allowance | Total_Salary |
+-------+------------+----------------------+--------+--------+-----------+--------------+
| 1     | John Smith | john.smith@email.com | M      | 50000  | 10000     | 60000        |
| 2     | Jane Doe   | jane.doe@email.com   | F      | 45000  | 12000     | 57000        |
+-------+------------+----------------------+--------+--------+-----------+--------------+
```

Explanation:

- It retrieves the employee's ID (`E_ID`), name (`E_Name`), email (`E_email`), and gender from the `EMPLOYEE` table.
- Then, it links each employee to their respective salary using the `EMPLOYEE_SALARY` table, which acts as a junction between employees and their salary records.
- The view also fetches the salary details like basic salary (`Basic`) and allowances (`Allowance`) from the `SALARY` table.
- Additionally, it computes the **total salary** by summing the basic salary and allowances for each employee.

2. View to Display Employee's Leave Details

```
CREATE VIEW EmployeeLeaveDetails AS
SELECT
    e.E_ID,
    e.E_Name,
    l.L_ID,
    l.Month
FROM
    EMPLOYEE e
JOIN
    `LEAVE` l ON e.E_ID = l.E_ID;
```

```
mysql> SELECT * FROM EmployeeLeaveDetails;
+-------+-------------+-------+--------+
| E_ID  | E_Name      | L_ID  | Month  |
+-------+-------------+-------+--------+
| 1     | John Smith  | 101   | Jan    |
| 2     | Jane Doe    | 102   | Feb    |
+-------+-------------+-------+--------+
```

Explanation:

- It selects employee details such as employee ID (`E_ID`) and name (`E_Name`) from the `EMPLOYEE` table.
- It joins this data with the `LEAVE` table to display the leave ID (`L_ID`) and the month in which the leave was taken (`Month`).
- The `LEAVE` table contains information about the leave taken by employees, and the view helps consolidate which employees took leave and in which month.

Enhanced ER Diagram

The **Enhanced Entity-Relationship (EER)** diagram is an advanced version of the standard ER diagram, used to model more complex relationships and data structures in a database. It extends the basic ER model by introducing additional concepts such as specialization, generalization, and inheritance, which are essential for capturing detailed and intricate relationships between entities in the salary management system.

In the context of a **Salary Management System**, the EER diagram helps in representing hierarchical relationships, such as differentiating between different types of employees (e.g., regular employees, managers, and contract employees) or categorizing financial transactions based on various attributes like date, amount, or month of salary payment.

By incorporating these advanced modeling techniques, the **EER diagram** allows for greater flexibility and clarity in the database design. It helps database designers manage real-world complexities such as hierarchical relationships in employee roles, salary structures, and tracking employee leave and transaction history. This added level of detail ensures that the data model is robust, scalable, and capable of handling the evolving needs of the salary management system, while also improving data accuracy and integrity.

The **EER diagram** serves as a critical tool in ensuring that all nuanced business requirements of the salary system are properly represented and supported in the database structure.

Key EER Concepts in the Salary Management System:

1. **Inheritance and Specialization**:
   - The **Employee** table acts as a generalized entity, while different types of employees, such as **Full-Time Employees** and **Part-Time Employees**, inherit attributes from the `Employee` entity.
   - For instance, **Full-Time Employees** might have additional attributes such as `Bonus` or `Retirement Funds`, while **Part-Time Employees** could have different salary structures with fewer benefits.
2. **Complex Relationships**:
   - **Salary Payments** involve multiple entities, including employees and salary details. The relationship between employees and their salary is captured through a **junction table (EMPLOYEE_SALARY)**, representing which employee is assigned which salary package.
   - **Leave Records** are associated with employees through the `LEAVE` table, and each leave entry indicates the specific month of leave taken.
   - **Transaction History** is related to both employees and their salary payments, with each transaction capturing details like payment date and amount, helping track the monthly salary disbursements for each employee.
3. **Advanced Entity Grouping**:
   - Employees might be further categorized based on their roles (e.g., **Manager**, **HR Staff**, **Finance Staff**), with specific attributes unique to each role. This helps in defining employee responsibilities, tracking role-specific data, and applying unique salary rules to different roles.

Benefits of the EER Diagram for the Salary Management System:

- **Enhanced Flexibility**: By using inheritance and specialization, it becomes easier to adapt the database for different types of employees and evolving business rules.
- **Improved Data Integrity**: The detailed relationships between entities ensure accurate salary tracking, leave management, and transaction history.
- **Scalability**: As the business grows or new employee types and salary structures are introduced, the database can easily be extended without major redesigns.

Constraints and Relationships

In a **salary management system**, constraints and relationships are essential for maintaining data accuracy and integrity. Constraints such as **primary keys, foreign keys, unique constraints**, and **NOT NULL** ensure that data like employee IDs, salary amounts, and transaction records are unique, valid, and complete. These rules prevent duplication and ensure that essential fields, such as salary amounts and employee details, are always filled. For example, each salary record must be linked to a valid employee, and every transaction must have a valid date and amount.

Relationships define how entities such as **employees, salaries, leave records, and transactions** are connected. These relationships ensure that data flows correctly between tables, allowing the system to track interactions like salary disbursements, leave tracking, and employee payments. By enforcing these connections, the database ensures accurate and efficient data retrieval, upholding the business logic and maintaining the integrity of salary operations.

VII. Purpose Of This Project

The purpose of the **Salary Management System** project is to streamline and automate the process of managing employee salaries, leave records, and payment transactions within an organization. By developing a structured database, the system ensures accurate salary calculations, proper linkage between employees and their respective salary details, and efficient tracking of payment history and leaves taken. The system aims to improve data accuracy, minimize errors, and provide a user-friendly platform for handling payroll operations, ultimately enhancing the organization's overall payroll management process.

VIII. Conclusion

The **Salary Management System** provides an efficient, reliable solution for managing payroll operations by automating salary calculations, leave tracking, and transaction recording. Through the use of a well-structured database with defined constraints and relationships, the system ensures data accuracy, integrity, and seamless retrieval of employee salary details. By reducing manual errors and improving the flow of payroll processes, this system supports smoother operations and helps organizations maintain consistent and transparent payroll management.