



EMPLOYEE PAYROLL SYSTEM

Project Report

BY
INSHA AFZAL
December 16, 2024

Project Title:

Employee Payroll System

Problem Statement:

The Employee Payroll System is designed to manage and process employee-related data, including salaries, leave records, employee information, and department details. The system must ensure accuracy in salary calculations, leave management, and data organization, all while maintaining data integrity and minimizing redundancy

Note:

For our project Employee Payroll System we have chosen 150 records of employees.

Entities Chosen:

The following entities were chosen to represent different aspects of the Employee Payroll System:

Employee_new

Payment

Leave Records

Salary Slip

Department

Employee

- EMPLOYEE_ID (Primary Key)
- EMAIL
- NAME
- ADDRESS
- PHONE
- DATE_OF_JOINING

Payment

- PAYMENT_ID (Primary Key)
- MONTH
- YEAR
- TOTAL_PAY

- BASIC_SALARY
- EMPLOYEE_ID (Foreign Key referencing Employee)

Leave Records

- LEAVE_ID (Primary Key)
- EMPLOYEE_ID (Foreign Key referencing Employee)
- LEAVE_TYPE
- LEAVE_DAYS
- LEAVE_DATE

Salary Slip

- SLIP_ID (Primary Key)
- EMPLOYEE_ID (Foreign Key referencing Employee)
- LEAVE_TYPE
- LEAVE_DATE
- LEAVE_DAYS

Department

- DEPARTMENT_ID (Primary Key)
- DEPARTMENT_NAME
- EMPLOYEE_ID (Foreign Key referencing Employee)

Data Model:

ER Model

The (ER Model) defines the relationships between entities. These entities are connected by relationships such as Employee to Payment, Employee to Leave Records, and Employee to Department.

For individual entity:

Employee_New	
Employee_ID 🔗	int
Email	varchar
Name	varchar
Address	varchar
Phone	varchar
Date_Of_Joining	date

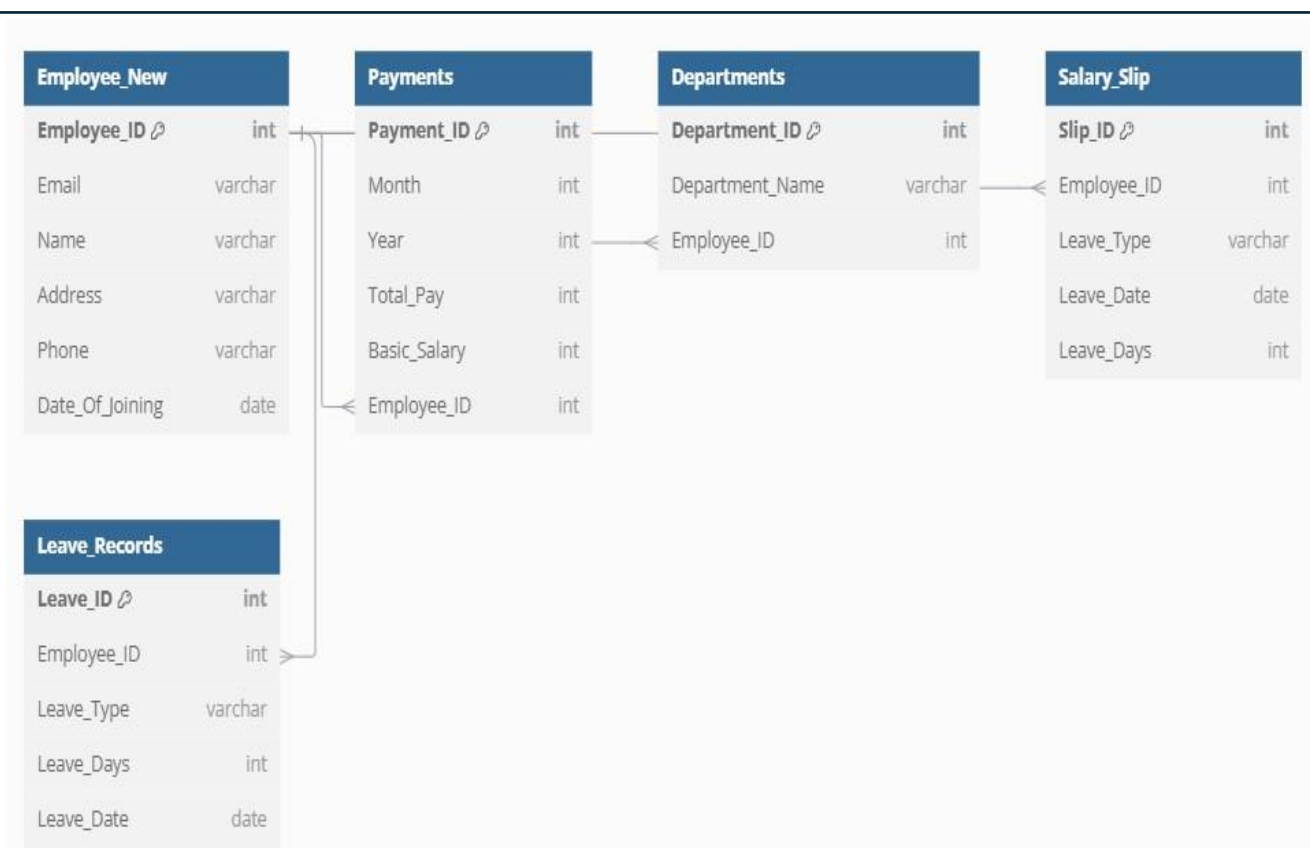
Payments	
Payment_ID 🔗	int
Month	int
Year	int
Total_Pay	int
Basic_Salary	int
Employee_ID	int

Leave_Records	
Leave_ID 🔗	int
Employee_ID	int
Leave_Type	varchar
Leave_Days	int
Leave_Date	date

Salary_Slip	
Slip_ID 🔗	int
Employee_ID	int
Leave_Type	varchar
Leave_Date	date
Leave_Days	int

Departments	
Department_ID 🔗	int
Department_Name	varchar
Employee_ID	int

Combined ER model:

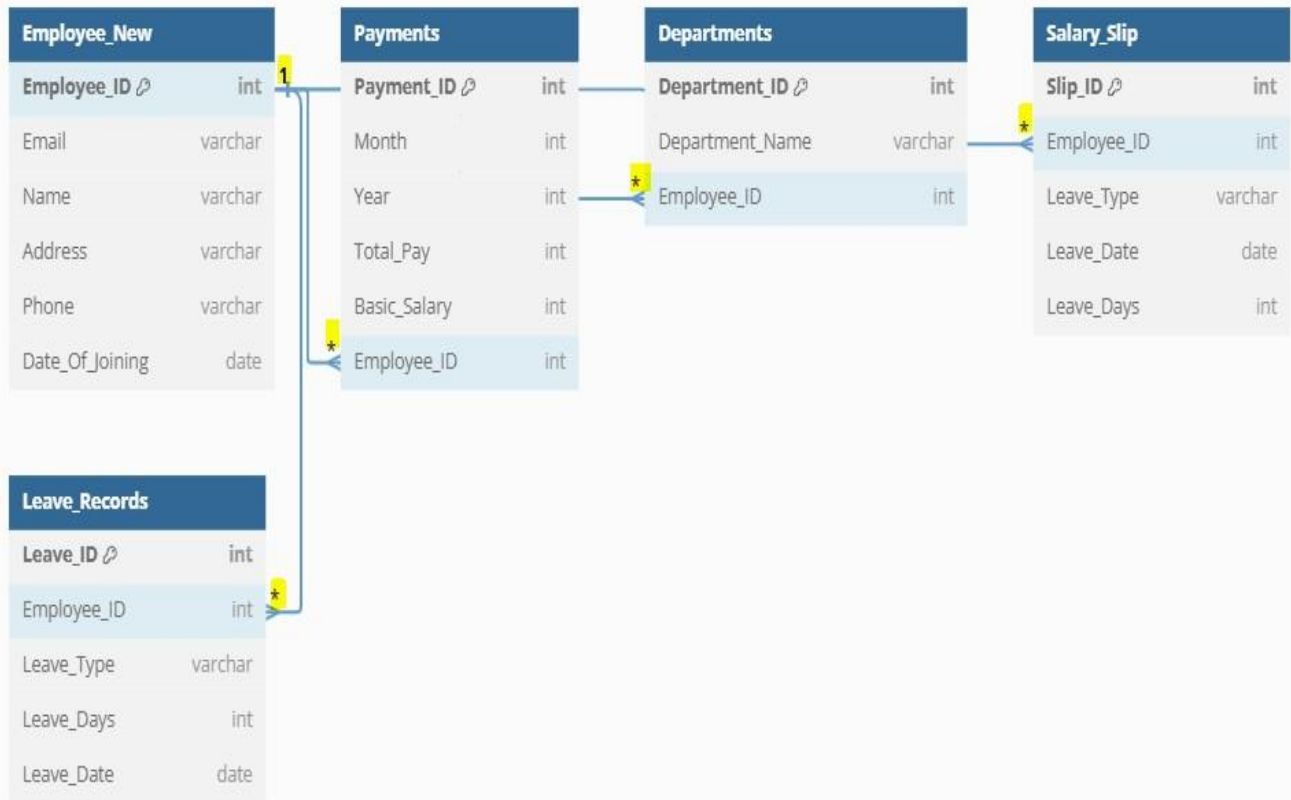


Cardnality:

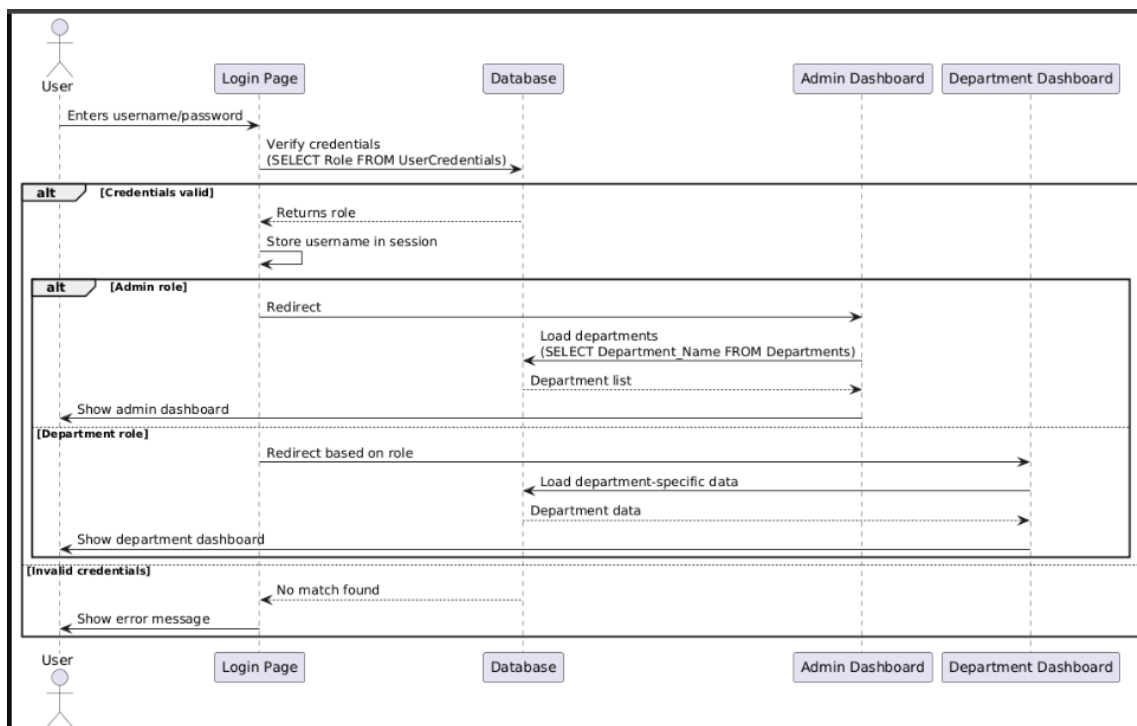
Here the relationship between entities is highlighted,

- ❖ Employee_New to Payments (Direct One-to-Many Relationship)
- ❖ Employee_New to Departments (One-to-One or One-to-Many Relationship)
- ❖ Employee_New to Salary_Slip (Direct One-to-Many Relationship)
- ❖ Employee_New to Leave_Records (Direct One-to-Many Relationship)
- ❖ Indirect Relationship Between Leave_Records and Salary_Slip(using Employee_New)

Graphically:



Sequence Diagram:



ii) Relational Model

The Relational Model is used to define how the data is organized into tables and how these tables are related to each other via primary and foreign keys

Relational model:

Using SQL:

Employee_New;

```
SQL> describe employee_new;
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(38)
EMAIL	NOT NULL	VARCHAR2(100)
NAME	NOT NULL	VARCHAR2(100)
ADDRESS	NOT NULL	VARCHAR2(255)
PHONE	NOT NULL	VARCHAR2(15)
DATE_OF_JOINING	NOT NULL	DATE

Payments:

Name	Null?	Type
PAYMENT_ID	NOT NULL	NUMBER(38)
MONTH	NOT NULL	VARCHAR2(15)
YEAR	NOT NULL	NUMBER(38)
TOTAL_PAY	NOT NULL	NUMBER(10,2)
BASIC_SALARY	NOT NULL	NUMBER(10,2)
EMPLOYEE_ID	NOT NULL	NUMBER(38)

Departments:

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(38)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(100)
EMPLOYEE_ID	NOT NULL	NUMBER(38)

Salary_slip:

Name	Null?	Type
SLIP_ID	NOT NULL	NUMBER(38)
EMPLOYEE_ID	NOT NULL	NUMBER(38)
LEAVE_TYPE	NOT NULL	VARCHAR2(50)
LEAVE_DATE	NOT NULL	DATE
LEAVE_DAYS	NOT NULL	NUMBER(38)

Leave_Records_:

Name	Null?	Type
LEAVE_ID	NOT NULL	NUMBER(38)
EMPLOYEE_ID	NOT NULL	NUMBER(38)
LEAVE_TYPE	NOT NULL	VARCHAR2(50)
LEAVE_DAYS	NOT NULL	NUMBER(38)
LEAVE_DATE	NOT NULL	DATE

iii) **Composite Model:** In this project, we have applied the **composite key** approach where two or more columns together uniquely identify a record. This is particularly useful in cases where a single attribute is not sufficient to ensure uniqueness. For instance:

- In the **Payments** table, the combination of Employee_ID and Month serves as a composite primary key to uniquely identify each payment record for an employee within a specific month.

- Similarly, in the **Leave_Records** table, Employee_ID and Leave_Date form a composite key, ensuring each leave record is uniquely associated with an employee and a specific leave date.

By using composite keys, we ensure data integrity and eliminate the need for additional surrogate keys, providing a more meaningful and efficient design for relationships between entities

Payments Table:

Purpose: To store payment details for employees.

Composite Key:

Employee_ID: Identifies the employee receiving the payment.

Month: Represents the month the payment was issued.

Composite Key Relationship:

(Employee_ID + Month) ensures that each employee's payment record for a specific month is unique.

Employee_ID	Month	Payment_Amount
101	January 2024	50,000
101	February 2024	50,000

Leave_Records Table

Purpose: To store employee leave details.

Composite Key:

Employee_ID: Identifies the employee taking the leave.

Leave_Date: The date of the leave.

Composite Key Relationship:

(Employee_ID + Leave_Date) ensures each employee's leave record is unique for a specific date

Employee_ID	Leave_Date	Reason
102	2024-03-15	Sick Leave
102	2024-03-16	Personal Leave

Normalized Schema:

Employee_New table: we took all atomic values for our project for “Employee_New table”

EMPLOYEE_ID	EMAIL	NAME	ADDRESS	PHONE	DATE_OF_JOINING
101	john@example.com	John Doe	123 Main St	123-456-7890	2021-05-01
102	jane@example.com	Jane Smith	456 Oak St	987-654-3210	2020-08-15

First Normal Form (1NF):

The table is already in **1NF** because all values are atomic and there are no repeating groups.

Second Normal Form (2NF)

The table is already in **2NF** because there is no partial dependency, and all non-key attributes depend on the entire primary key (EMPLOYEE_ID)

Third Normal Form (3NF)

The table is in **3NF** because we removed any potential transitive dependency by separating CITY and STATE into a new City table, and now CITY_ID is used in the Employee table.

Final Conclusion for Your Project Regarding Normalization:

In the same manner, the data in the Employee Payroll System has been cleaned and normalized through 1NF, 2NF, and 3NF. We ensured that all data is atomic, removed partial dependencies, and eliminated any transitive dependencies to create a well-organized, efficient table structure. This approach helped streamline the data for accurate and efficient payroll processing.

SQL Commands:

Tables Creation:

```

SQL> BEGIN
2  -- Optional: Delete existing records if the table needs to be reset
3  DELETE FROM Employee_New;
4  COMMIT;
5
6  -- Insert loop for 150 records
7  FOR i IN 1..150 LOOP
8      INSERT INTO Employee_New (Employee_ID, Email, Name, Address, Phone, Date_Of_Joining)
9      VALUES (
10         i, -- Employee_ID
11         'employee'||i||'@example.com', -- Email
12         'Employee '||i, -- Name
13         'Address '||i, -- Address
14         '0300123'||LPAD(i, 4, '0'), -- Phone
15         TO_DATE('2020-01-01', 'YYYY-MM-DD') + MOD(i, 365) -- Date_Of_Joining
16     );
17 END LOOP;
18 COMMIT;
19 DBMS_OUTPUT.PUT_LINE('150 records inserted successfully into Employee_New.');
```

```

SQL> BEGIN
2  FOR i IN 1..150 LOOP
3      INSERT INTO Payments (Payment_ID, Month, Year, Total_Pay, Basic_Salary, Employee_ID)
4      VALUES (
5         i, -- Payment_ID
6         MOD(i, 12) + 1, -- Month (1-12)
7         2020, -- Year
8         50000 + MOD(i, 5000), -- Total Pay
9         30000 + MOD(i, 3000), -- Basic Salary
10        i -- Employee_ID (foreign key from Employee_New)
11    );
12 END LOOP;
13 COMMIT;
14 DBMS_OUTPUT.PUT_LINE('150 records inserted successfully into Payments.');
```

PL/SQL procedure successfully completed.

```

SQL> BEGIN
2  FOR i IN 1..150 LOOP
3      INSERT INTO Departments (Department_ID, Department_Name, Employee_ID)
4      VALUES (
5         i, -- Department_ID
6         'Department ' || MOD(i, 5) + 1, -- Department Name (5 departments)
7         i -- Employee_ID (foreign key from Employee_New)
8     );
9 END LOOP;
10 COMMIT;
11 DBMS_OUTPUT.PUT_LINE('150 records inserted successfully into Departments.');
```

In similier manner like above we created as well as inserted data in all tables for our project, here we have used automatic approach for inserting records for 150 employees , we can insert these manually.

Testing:

Run sql queries for testing :

1. Simple Query to Check the Total Number of Employees (150 records):

```
SQL> SELECT COUNT(*)
2 FROM Employee_New;

COUNT(*)
-----
150
```

2. Check the Names of the First 5 Employees:

```
SQL> SELECT Name
2 FROM Employee_New
3 WHERE ROWNUM <= 5;

NAME
-----
Employee 1
Employee 2
Employee 3
Employee 4
Employee 5
```

3. Check the Salary Slip of the First 3 Employees

```
SQL> SELECT Payment_ID, Employee_ID, Total_Pay, Basic_Salary
2 FROM Payments
3 WHERE Employee_ID <= 3;

PAYMENT_ID EMPLOYEE_ID TOTAL_PAY BASIC_SALARY
-----
1          1          50001      30001
2          2          50002      30002
3          3          50003      30003
```

4. Employee, Payments, and Salary_Slip for Employees 1 to 3

```
SQL> SELECT e.Name, p.Total_Pay, s.Leave_Type, s.Leave_Days
  2  FROM Employee_New e
  3  JOIN Payments p ON e.Employee_ID = p.Employee_ID
  4  JOIN Salary_Slip s ON e.Employee_ID = s.Employee_ID
  5  WHERE e.Employee_ID BETWEEN 1 AND 3;
```

NAME

	TOTAL_PAY	LEAVE_TYPE	LEAVE_DAYS
Employee 1	50001	Casual	1
Employee 2	50002	Sick	2
Employee 3	50003	Casual	3

5. Get Employees with Total Pay equal to 50138

```
SQL> SELECT e.Name, SUM(p.Total_Pay) AS Total_Pay
  2  FROM Employee_New e
  3  JOIN Payments p ON e.Employee_ID = p.Employee_ID
  4  GROUP BY e.Name
  5  HAVING SUM(p.Total_Pay) = 50138;
```

NAME

	TOTAL_PAY
Employee 138	50138

6. The query returns the total payment per month in ascending order of the month number.

```
SQL> SELECT p.Month, SUM(p.Total_Pay) AS Total_Pay_Month
  2 FROM Payments p
  3 GROUP BY p.Month
  4 ORDER BY TO_NUMBER(p.Month);
```

MONTH	TOTAL_PAY_MONTH
1	600936
2	650949
3	650962
4	650975
5	650988
6	651001
7	651014
8	600876
9	600888
10	600900
11	600912

MONTH	TOTAL_PAY_MONTH
12	600924

12 rows selected.

Manually we want to add new employee:

Insert a new employee named "Insha" into all the relevant tables

```
SQL> INSERT INTO Employee_New (Employee_ID, Email, Name, Address, Phone, Date_Of_Joining)
  2 VALUES (151, 'insha11233@...', 'Insha', 'Khanpur', '03005678901', TO_DATE('2024-12-11', 'YYYY-MM-DD'));

1 row created.

SQL> INSERT INTO Payments (Payment_ID, Month, Year, Total_Pay, Basic_Salary, Employee_ID)
  2 VALUES (151, '12', 2024, 50000, 30000, 151);

1 row created.

SQL> INSERT INTO Departments (Department_ID, Department_Name, Employee_ID)
  2 VALUES (151, 'Department 1', 151);

1 row created.

SQL> INSERT INTO Salary_Slip (Slip_ID, Employee_ID, Leave_Type, Leave_Date, Leave_Days)
  2 VALUES (151, 151, 'Sick Leave', TO_DATE('2024-12-11', 'YYYY-MM-DD'), 1);

1 row created.

SQL> INSERT INTO Leave_Records (Leave_ID, Employee_ID, Leave_Type, Leave_Days, Leave_Date)
  2 VALUES (151, 151, 'Annual Leave', 1, TO_DATE('2024-12-11', 'YYYY-MM-DD'));

1 row created.
```

Result:

7. To view only the name, phone number, and city (address) of the employee "Insha" from the Employee_New

```
SQL> SELECT Name, Phone, Address
2 FROM Employee_New
3 WHERE Name = 'Insha';
```

NAME

PHONE

ADDRESS

Insha

03005678901

Khanpur

Triggers:

DELETE trigger for the Leave_Records table. This trigger prevents any record from being deleted from the table by raising an error:

```
SQL> CREATE OR REPLACE TRIGGER trg_prevent_delete
2 BEFORE DELETE ON Leave_Records
3 FOR EACH ROW
4 BEGIN
5     RAISE_APPLICATION_ERROR(-20002, 'Deletion of leave records is not allowed.');
```

Trigger created.

```
SQL> DELETE FROM Leave_Records WHERE Leave_ID = 151;
DELETE FROM Leave_Records WHERE Leave_ID = 151
*
```

ERROR at line 1:

ORA-20002: Deletion of leave records is not allowed.

ORA-20002: Deletion of leave records is not allowed.

Trigger for Validating Leave Days in Leave_Records: Prevents employees from taking more than 5 leave days at a time.

```
SQL> CREATE OR REPLACE TRIGGER trg_validate_leave_days
  2 BEFORE INSERT OR UPDATE ON Leave_Records
  3 FOR EACH ROW
  4 BEGIN
  5     IF :NEW.Leave_Days > 5 THEN
  6         RAISE_APPLICATION_ERROR(-20001, 'Leave days cannot exceed 5.');
```

Trigger created.

Output:

```
SQL> INSERT INTO Leave_Records (Leave_ID, Employee_ID, Leave_Type, Leave_Days, Leave_Date)
  2 VALUES (152, 1, 'Casual', 6, TO_DATE('2024-06-01', 'YYYY-MM-DD'));
INSERT INTO Leave_Records (Leave_ID, Employee_ID, Leave_Type, Leave_Days, Leave_Date)
*
ERROR at line 1:
ORA-20001: Leave days cannot exceed 5.
ORA-06543: at "HR.TRG_VALIDATE_LEAVE_DAYS", line 2
```

Size Estimation of

- Record
- Table
- DB

Employee_New Table:

EMPLOYEE_ID: NUMBER(38) = 38 bytes

EMAIL: VARCHAR2(100) = 100 bytes

NAME: VARCHAR2(100) = 100 bytes

ADDRESS: VARCHAR2(255) = 255 bytes

PHONE: VARCHAR2(15) = 15 bytes

DATE_OF_JOINING: DATE = 7 bytes

Total size for one record in Employee_New = 38 + 100 + 100 + 255 + 15 + 7 = 515 bytes

Payments Table:

PAYMENT_ID: NUMBER(38) = 38 bytes

MONTH: VARCHAR2(15) = 15 bytes

YEAR: NUMBER(38) = 38 bytes

TOTAL_PAY: NUMBER(10,2) = 10 bytes

BASIC_SALARY: NUMBER(10,2) = 10 bytes

EMPLOYEE_ID: NUMBER(38) = 38 bytes

Total size for one record in Payments = $38 + 15 + 38 + 10 + 10 + 38 = 146$ bytes

Leave_Records Table:

LEAVE_ID: NUMBER(38) = 38 bytes

EMPLOYEE_ID: NUMBER(38) = 38 bytes

LEAVE_TYPE: VARCHAR2(50) = 50 bytes

LEAVE_DAYS: NUMBER(38) = 38 bytes

LEAVE_DATE: DATE = 7 bytes

Total size for one record in Leave_Records = $38 + 38 + 50 + 38 + 7 = 171$ bytes

Salary_Slip Table:

SLIP_ID: NUMBER(38) = 38 bytes

EMPLOYEE_ID: NUMBER(38) = 38 bytes

LEAVE_TYPE: VARCHAR2(50) = 50 bytes

LEAVE_DATE: DATE = 7 bytes

LEAVE_DAYS: NUMBER(38) = 38 bytes

Total size for one record in Salary_Slip = $38 + 38 + 50 + 7 + 38 = 171$ bytes

Departments Table:

DEPARTMENT_ID: NUMBER(38) = 38 bytes

DEPARTMENT_NAME: VARCHAR2(100) = 100 bytes

EMPLOYEE_ID: NUMBER(38) = 38 bytes

Total size for one record in Departments = $38 + 100 + 38 = 176$ bytes

Estimate Total Size of Tables

Payments (150 records):

$149 \text{ bytes} \times 150 = 22,350 \text{ bytes} = 22.35 \text{ KB}$

Employee_New (150 records):

$515 \text{ bytes} \times 150 = 77,250 \text{ bytes} = 77.25 \text{ KB}$

Leave_Records (150 records):

$171 \text{ bytes} \times 150 = 25,650 \text{ bytes} = 25.65 \text{ KB}$

Salary_Slip (150 records):

$171 \text{ bytes} \times 150 = 25,650 \text{ bytes} = 25.65 \text{ KB}$

Departments (150 records):

$176 \text{ bytes} \times 150 = 26,400 \text{ bytes} = 26.4 \text{ KB}$

Total Database Size:

Total size of database = Payments size + Employee_New size + Leave_Records size + Salary_Slip size + Departments size

$= 22.35 \text{ KB} + 77.25 \text{ KB} + 25.65 \text{ KB} + 25.65 \text{ KB} + 26.4 \text{ KB}$

$= 177.3 \text{ KB}$

Final Summary:

In this project, we developed an Employee Payroll System to manage employee-related data, such as payments, leave records, and department details. The database was designed using an Entity-Relationship Model and translated into a Relational Model for better data management. We ensured the data was organized and efficient by normalizing the schema through 1NF, 2NF, and 3NF, eliminating redundancy and ensuring data integrity. SQL commands were used to create the database, insert sample data, and implement comprehensive queries and triggers for functionality. Finally, size estimations were performed for each table and record to understand the database's storage requirements. This approach guarantees a well-structured, optimized, and scalable system for managing employee payroll efficiently.