SYMBIOSIS INSTITUTE OF TECHNOLOGY, NAGPUR

MINI PROJECT REPORT
ON
"REAL ESTATE DATABASE MANAGEMENT SYSTEM"

B. TECH

COMPUTER SCIENCE AND ENGINEERING

BATCH: 2023-27

SESSION:2024-25

Course Name – Database Management System

Course Code - 0705210505

SEMESTER - 4

SECTION – B

Submitted by:

Name: Shantanu k.khope PRN: 23070521134

Name: Surajit Halder PRN: 23070521160

Table of Contents

1. Introduction

The **Hostel Management Database System** is developed to streamline and automate the various operations involved in managing hostel accommodations within a university or college environment. This system covers the management of students, rooms, wardens, visitors, fee payments, complaints, and room allotments. By leveraging a **relational database model**, the system ensures **data accuracy**, facilitates **efficient data retrieval**, and automates critical administrative processes. Advanced database concepts such as **SQL queries**, **PL/SQL procedures**, **functions**, and **triggers** are used to enhance system functionality, enforce data integrity, and simplify complex hostel operations.

2. Objectives

The primary objectives of this project are:

- **Database Design and Implementation**:
  To design a robust relational database that effectively manages hostel-related data, including students, rooms, wardens, visitors, fee payments, allotments, and complaints.

- **Data Integrity Enforcement**:
  To maintain data accuracy and consistency using SQL constraints such as primary keys, foreign keys, and check constraints.

- **Process Automation**:
  To automate essential administrative processes through the use of PL/SQL functions, procedures, and triggers, thereby improving efficiency and reducing manual intervention.

- **Efficient Data Retrieval**:
  To develop advanced SQL queries that support reporting, monitoring of room availability, fee tracking, and complaint resolution.

- **Performance Optimization**:
  To improve overall database performance using indexing, normalization, and optimized query structures.

3. Theoretical Framework

3.1. Relational Database Model

Theory:
The **relational database model** organizes data into structured tables (relations), where each table consists of rows and columns. Each table represents a specific entity, and the relationships between these entities are established using keys such as **primary keys** and **foreign keys**. This model ensures **data consistency**, **referential integrity**, and supports data operations using **Structured Query Language (SQL)**.

Project Example:
In the **Hostel Management System**, entities such as **Student**, **Room**, **Warden**, **Visitor**, **FeePayment**, **Complaint**, and **Allotment** are represented as individual tables. The attributes of each entity are stored as columns within their respective tables. Relationships—such as a student being allotted a room, raising a complaint, or receiving a visitor—are maintained through foreign key associations, ensuring a structured and logically connected data environment.

3.2. Entity-Relationship (ER) Modeling

Theory:
ER modeling is a conceptual design approach used to represent the logical structure of data and the relationships between different data elements in a system. It uses **entities**, **attributes**, and **relationships** to visually illustrate how information is organized and interconnected.

Project Example:
The ER diagram for the **Hostel Management System** includes

entities such as **Student**, **Room**, **Warden**, **Visitor**, **Allotment**, **FeePayment**, and **Complaint**. For example, a **Student** is related to a **Room** through the **Allotment** entity, and can also raise a **Complaint** or receive a **Visitor**. These relationships help to clearly define the interactions within the hostel ecosystem.

🔷 3.3. SQL Constraints

Theory:
SQL constraints are rules applied to table columns to ensure **data integrity and accuracy**. These constraints help maintain the quality of the data stored in the database.

- **Primary Key:** Uniquely identifies each record in a table.
- **Foreign Key:** Maintains referential integrity between related tables.
- **Unique:** Ensures that all values in a column are different.
- **Not Null:** Ensures a column cannot store NULL values.
- **Check:** Validates that values in a column meet specific conditions.

Project Example:
In the Hostel Management System, the **StudentID** in the `Student` table is a **Primary Key**, while it appears as a **Foreign Key** in the `Allotment`, `FeePayment`, and `Complaint` tables. A **Check Constraint** is used in the `Room` table to ensure the room capacity does not exceed allowed limits.

🔷 3.4. SQL Joins

Theory:
SQL Joins are used to **combine data from two or more tables** based on related columns. They are essential for querying data from normalized databases.

- **INNER JOIN:** Returns only matching rows from both tables.
- **LEFT JOIN:** Returns all rows from the left table and matched rows from the right.
- **RIGHT JOIN:** Returns all rows from the right table and matched rows from the left.
- **FULL JOIN:** Returns all rows when there is a match in either table.

Project Example:
To retrieve all students and their room numbers, an **INNER JOIN** is used between the `Student` and `Allotment` tables. Similarly, a **LEFT JOIN** can be used to list students even if they haven't raised a complaint.

🔷 3.5. PL/SQL Functions and Procedures

Theory:
PL/SQL is an extension of SQL that allows writing procedural code with **control structures** like loops, conditionals, and exception handling.

- A **Function** returns a value.

- A **Procedure** performs operations but does not return a value directly.

Project Example:
A procedure named `auto_assign_room` can be created to automatically allot the first available room to a student. A function like `calculate_pending_fees(StudentID)` can return the pending fee amount for a specific student.

🔷 3.6. Triggers

Theory:
Triggers are special PL/SQL procedures that **execute automatically** in response to **INSERT**, **UPDATE**, or **DELETE** operations on a table. They help in enforcing business logic and maintaining audit trails.

Project Example:
A trigger named `prevent_room_overbooking` can be used to ensure that a room's capacity is not exceeded when assigning students. Another trigger can automatically mark a complaint as "Pending" when inserted.

3.7. Views

Theory:
A **View** is a virtual table based on the result of a query. Views simplify complex SQL logic, restrict access to sensitive data, and present data in a consistent and readable format.

Project Example:
A view named `student_room_view` can be created to show student names along with their room details. Another view,

`pending_complaints_view`, can list all unresolved complaints for easy monitoring by hostel authorities.

4. ER Diagram Explanation

The ER diagram represents the entities and their relationships within the Hostel Management system:

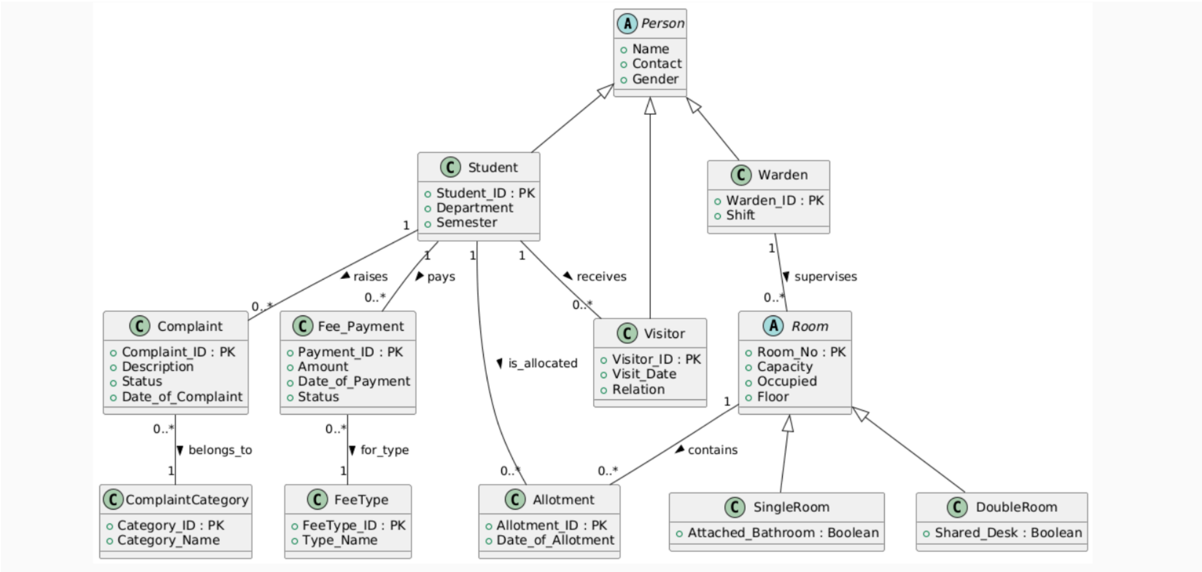


Fig: Entity-Relationship Model

- **Student**:
  Represents students residing in the hostel, with attributes such as `StudentID`, `Name`, `Email`, `Gender`, `PhoneNo`, and `Department`.

- **Room**:
  Represents hostel rooms available for allotment, with attributes like `RoomNo`, `Capacity`, `Floor`, and `RoomType` (e.g., Single, Double).

- **Warden**:
  Represents hostel wardens responsible for room management and student coordination. Attributes include `WardenID`, `Name`, `Contact`, and `Shift`.

- **Visitor**:
  Represents individuals visiting students in the hostel. Includes attributes such as `VisitorID`, `Name`, `VisitDate`, `Purpose`, and `Relation`.

- **Allotment**:
  Represents the assignment of students to specific rooms, with attributes like `AllotmentID`, `StudentID`, `RoomNo`, and `DateOfAllotment`.

- **FeePayment**:
  Records payments made by students towards hostel fees. Includes attributes like `PaymentID`, `StudentID`, `Amount`, `Date`, and `Status`.

- **Complaint**:
  Represents complaints raised by students, with attributes such as `ComplaintID`, `StudentID`, `Category`, `Description`, `Date`, and `Status`.

- **ComplaintCategory**:
  Categorizes types of complaints (e.g., Maintenance, Cleanliness), with attributes like `CategoryID` and `CategoryName`.

- **Visit**:
  Maps visitors to students, tracking the relationship and date of visit. Includes `VisitID`, `VisitorID`, and `StudentID`.

Relationships:

Relationships in the Hostel Management System

- A **Student** can be **allotted one room** at a time, but a **Room** can be **shared by multiple students** depending on its capacity.

- A **Student** can raise **multiple Complaints**, but each **Complaint** is linked to **only one Student** and categorized under a specific **ComplaintCategory**.

- A **Warden** can be responsible for **multiple Rooms**, but each **Room** is managed by a **single Warden**.

- A **Student** can have **multiple FeePayment records**, and each **FeePayment** is linked to **one Student**.

- A **Student** can receive **multiple Visitors**, and each **Visitor** visit is linked to **a specific Student** via the **Visit** relationship.

- An **Allotment** links a **Student** to a **Room**, tracking the **date** of allocation and the specific **room number** allotted.

5.Database Schema Design

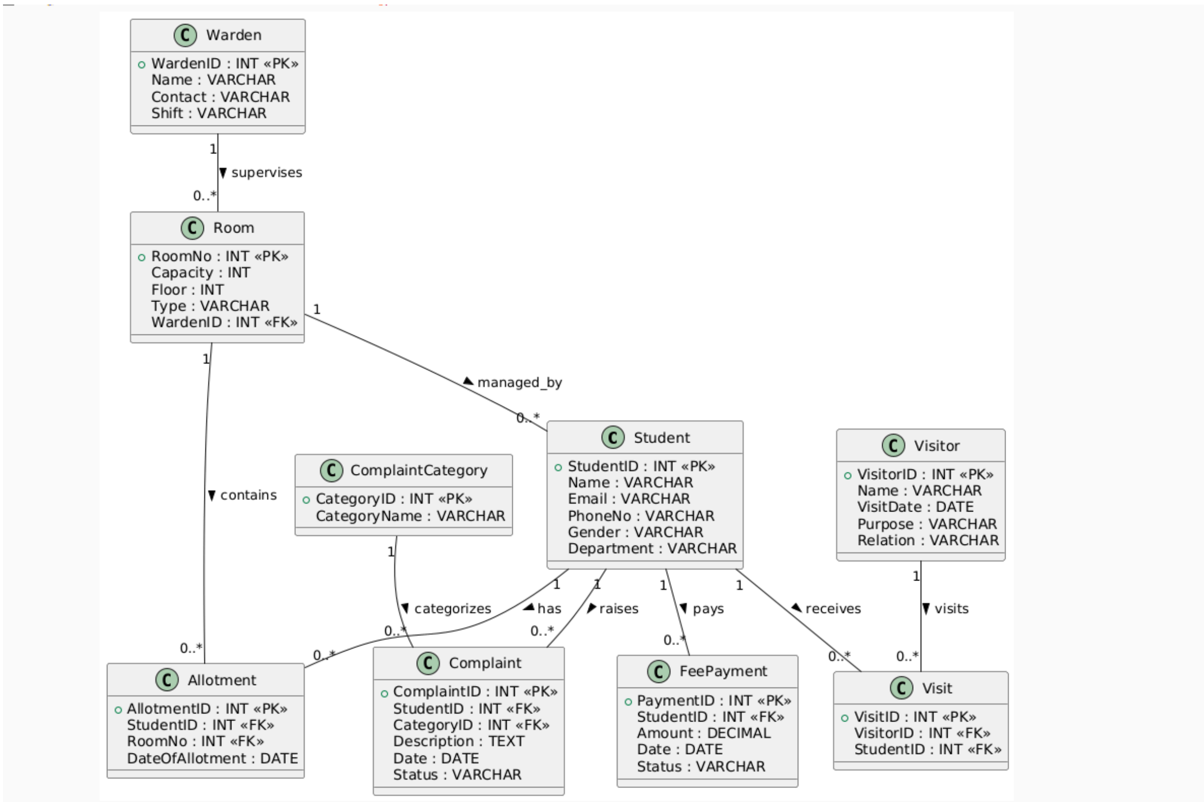


Fig: Schema Diagram

5.1. Tables and Relationships

1. **HouseOwners**
   - **Attributes**: CustomerID, HouseID
   - **Description**: Associates customers with the houses they own.
2. **House**
   - **Attributes**: HouseID, HouseTypeID, Address, Neighborhood, City, Owned, SqrFt, CustomerID
   - **Description**: Stores details of each house.
3. **HouseType**
   - **Attributes**: HouseTypeID, TypeName
   - **Description**: Defines different types of houses.
4. **Customers**
   - **Attributes**: CustomerID, Name, Email, PhoneNum, Owner
   - **Description**: Stores customer information.
5. **Sale**
   - **Attributes**: SaleID, SalesManID, HouseID, SellerID, BuyerID, SalePrice, SaleDate, AskedPrice
   - **Description**: Records completed sales transactions.
6. **SalesMan**
   - **Attributes**: SalesManID, Name, BirthDate, Email, PhoneNum
   - **Description**: Stores information about salesmen.
7. **Offer**
   - **Attributes**: OfferID, HouseID, CustomerID, SalesManID, OfferPrice, OfferDate
   - **Description**: Manages offers made by customers.
8. **SmanExpertise**
   - **Attributes**: SalesManID, HouseTypeID
   - **Description**: Maps salesmen to their areas of expertise.

5.2. Relationships

- **HouseOwners**: Many-to-Many relationship between **Customers** and **House**.
- **House** to **HouseType**: Many-to-One relationship.
- **Sale**: Links **SalesMan**, **House**, **Seller (Customer)**, and **Buyer (Customer)**.
- **Offer**: Links **House**, **Customer**, and **SalesMan**.
- **SmanExpertise**: Many-to-Many relationship between **SalesMan** and **HouseType**

6. Implementation

6.1. Table Creation & Data Insertion

Table 1: House

```
mysql> desc house;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
```

```
mysql> desc salesman;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| SalesManID | int          | NO   | PRI | NULL    |       |
| Name       | varchar(100) | YES  |     | NULL    |       |
| BirthDate  | date         | YES  |     | NULL    |       |
| Email      | varchar(100) | YES  |     | NULL    |       |
| PhoneNum   | varchar(15)  | YES  | UNI | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql> select * from salesman;
+------------+---------------+------------+--------------------------+------------+
| SalesManID | Name          | BirthDate  | Email                    | PhoneNum   |
+------------+---------------+------------+--------------------------+------------+
|        301 | Alice Johnson | 1985-05-15 | alice.johnson@gmail.com  | 9871234560 |
|        302 | Bob Williams  | 1990-08-22 | bob.williams@gmail.com   | 8765123490 |
+------------+---------------+------------+--------------------------+------------+
2 rows in set (0.02 sec)
```

Table 2: Salesman

```
mysql> desc offer;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| OfferID    | int           | NO   | PRI | NULL    |       |
| HouseID    | int           | YES  | MUL | NULL    |       |
| CustomerID | int           | YES  | MUL | NULL    |       |
| SalesManID | int           | YES  | MUL | NULL    |       |
| OfferPrice | decimal(10,2) | YES  |     | NULL    |       |
| OfferDate  | date          | YES  |     | NULL    |       |
+------------+---------------+------+-----+---------+-------+
6 rows in set (0.04 sec)

mysql> select * from offer;
+---------+---------+------------+------------+------------+------------+
| OfferID | HouseID | CustomerID | SalesManID | OfferPrice | OfferDate  |
+---------+---------+------------+------------+------------+------------+
|     501 |     201 |        102 |        301 |  240000.00 | 2023-12-25 |
|     502 |     202 |        103 |        302 |  190000.00 | 2024-01-10 |
|     503 |     203 |        104 |        301 |  340000.00 | 2024-02-05 |
|     504 |     204 |        105 |        302 |  410000.00 | 2024-03-18 |
|     505 |     205 |        106 |        302 |  510000.00 | 2024-04-20 |
|     506 |     206 |        107 |        301 |  620000.00 | 2024-05-15 |
|     507 |     207 |        101 |        302 |   94000.00 | 2024-06-01 |
```

Table 3: Offer

```
mysql> desc smanexpertise;
+-------------+------+------+-----+---------+-------+
| Field       | Type | Null | Key | Default | Extra |
+-------------+------+------+-----+---------+-------+
| SalesManID  | int  | NO   | PRI | NULL    |       |
| HouseTypeID | int  | NO   | PRI | NULL    |       |
+-------------+------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> select * from smanexpertise;
+------------+-------------+
| SalesManID | HouseTypeID |
+------------+-------------+
|        301 |           1 |
|        302 |           1 |
|        301 |           2 |
|        302 |           2 |
|        301 |           3 |
|        302 |           3 |
+------------+-------------+
6 rows in set (0.02 sec)
```

Table 4: Smanexpertise

```
mysql> desc housetype;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| HouseTypeID | int         | NO   | PRI | NULL    |       |
| TypeName    | varchar(50) | YES  | UNI | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> select * from housetype;
+-------------+-----------+
| HouseTypeID | TypeName  |
+-------------+-----------+
|           1 | Apartment |
|           2 | Bungalow  |
|           3 | Villa     |
+-------------+-----------+
3 rows in set (0.01 sec)
```

Table 5: HouseType

```
mysql> desc sale;
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| SaleID      | int           | NO   | PRI | NULL    |       |
| SalesManID  | int           | YES  | MUL | NULL    |       |
| HouseID     | int           | YES  | MUL | NULL    |       |
| SellerID    | int           | YES  | MUL | NULL    |       |
| BuyerID     | int           | YES  | MUL | NULL    |       |
| SalePrice   | decimal(10,2) | YES  |     | NULL    |       |
| SaleDate    | date          | YES  |     | NULL    |       |
| AskedPrice  | decimal(10,2) | YES  |     | NULL    |       |
+-------------+---------------+------+-----+---------+-------+
8 rows in set (0.00 sec)

mysql> select * from sale;
+--------+------------+---------+----------+---------+-----------+------------+------------+
| SaleID | SalesManID | HouseID | SellerID | BuyerID | SalePrice | SaleDate   | AskedPrice |
+--------+------------+---------+----------+---------+-----------+------------+------------+
|    401 |        301 |     201 |      101 |     102 | 250000.00 | 2024-01-15 |  255000.00 |
|    402 |        302 |     202 |      102 |     103 | 200000.00 | 2024-02-01 |  210000.00 |
|    403 |        301 |     203 |      103 |     104 | 350000.00 | 2024-03-10 |  360000.00 |
|    404 |        302 |     204 |      104 |     105 | 400000.00 | 2024-04-22 |  420000.00 |
|    405 |        302 |     205 |      105 |     106 | 500000.00 | 2024-05-30 |  510000.00 |
|    406 |        301 |     206 |      106 |     107 | 600000.00 | 2024-06-15 |  610000.00 |
|    407 |        302 |     207 |      107 |     101 |  90000.00 | 2024-07-01 |   95000.00 |
|    503 |        301 |     208 |      101 |     102 | 275000.00 | 2024-03-10 |  280000.00 |
+--------+------------+---------+----------+---------+-----------+------------+------------+
8 rows in set (0.00 sec)
```

Table 6: Sale

Table 7: Customers

```
mysql> desc customers;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| CustomerID | int          | NO   | PRI | NULL    |       |
| Name       | varchar(100) | YES  |     | NULL    |       |
| Email      | varchar(100) | NO   |     | NULL    |       |
| PhoneNum   | varchar(15)  | NO   | UNI | NULL    |       |
| Owner      | tinyint(1)   | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql> select * from customers;
+------------+-------------------+-------------------------+------------+-------+
| CustomerID | Name              | Email                   | PhoneNum   | Owner |
+------------+-------------------+-------------------------+------------+-------+
|        101 | John Doe          | john.doe@gmail.com      | 9876543210 |     1 |
|        102 | Jane Smith        | jane.smith@gmail.com    | 8765432109 |     0 |
|        103 | Michael Johnson   | michael.j@gmail.com     | 7654321098 |     1 |
|        104 | Emily Davis       | emily.davis@gmail.com   | 6543210987 |     1 |
|        105 | Christopher Moore | chris.moore@gmail.com   | 5432109876 |     0 |
|        106 | Olivia Brown      | olivia.brown@gmail.com  | 4321098765 |     1 |
|        107 | Liam Wilson       | liam.wilson@gmail.com   | 3210987654 |     0 |
+------------+-------------------+-------------------------+------------+-------+
7 rows in set (0.00 sec)
```

Table 8: HouseOwner

```
mysql> select * from HouseOwner;
+------------+---------+
| CustomerID | HouseID |
+------------+---------+
|        101 |     201 |
|        102 |     202 |
|        103 |     203 |
|        104 |     204 |
|        105 |     205 |
|        106 |     206 |
|        107 |     207 |
+------------+---------+
7 rows in set (0.02 sec)
```

6.2. Adding Constraints

After inserting data, it's essential to enforce data integrity through primary keys and foreign keys.

```
MySQL 8.0 Command Line Cli

mysql> ALTER TABLE House
    -> ADD CONSTRAINT fk_house_housetype
    -> FOREIGN KEY (HouseTypeID) REFERENCES HouseType(HouseTypeID),
    -> ADD CONSTRAINT fk_house_customer
    -> FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);
Query OK, 8 rows affected (0.17 sec)
Records: 8  Duplicates: 0  Warnings: 0

mysql>
mysql> -- Foreign Keys for Sale
mysql> ALTER TABLE Sale
```

Fig: Constraints

```
MySQL 8.0 Command Line Cli

mysql> -- Add Primary Keys
mysql> ALTER TABLE HouseOwners
    -> ADD PRIMARY KEY (CustomerID, HouseID);
ERROR 1146 (42S02): Table 'realestate.houseowners' doesn't exist
mysql>
mysql> ALTER TABLE House
    -> ADD PRIMARY KEY (HouseID);
ERROR 1068 (42000): Multiple primary key defined
mysql>
mysql> ALTER TABLE HouseType
    -> ADD PRIMARY KEY (HouseTypeID);
ERROR 1068 (42000): Multiple primary key defined
mysql>
mysql> ALTER TABLE Customers
    -> ADD PRIMARY KEY (CustomerID);
ERROR 1068 (42000): Multiple primary key defined
mysql>
mysql> ALTER TABLE Sale
    -> ADD PRIMARY KEY (SaleID);
ERROR 1068 (42000): Multiple primary key defined
mysql>
```

Fig: Constraints

Explanation:

- **Primary Keys** ensure each record in a table is unique.
- **Foreign Keys** enforce referential integrity by linking related records across tables.

6.3. Advanced SQL Queries:

Retrieve Salesman Performance Based on Sales

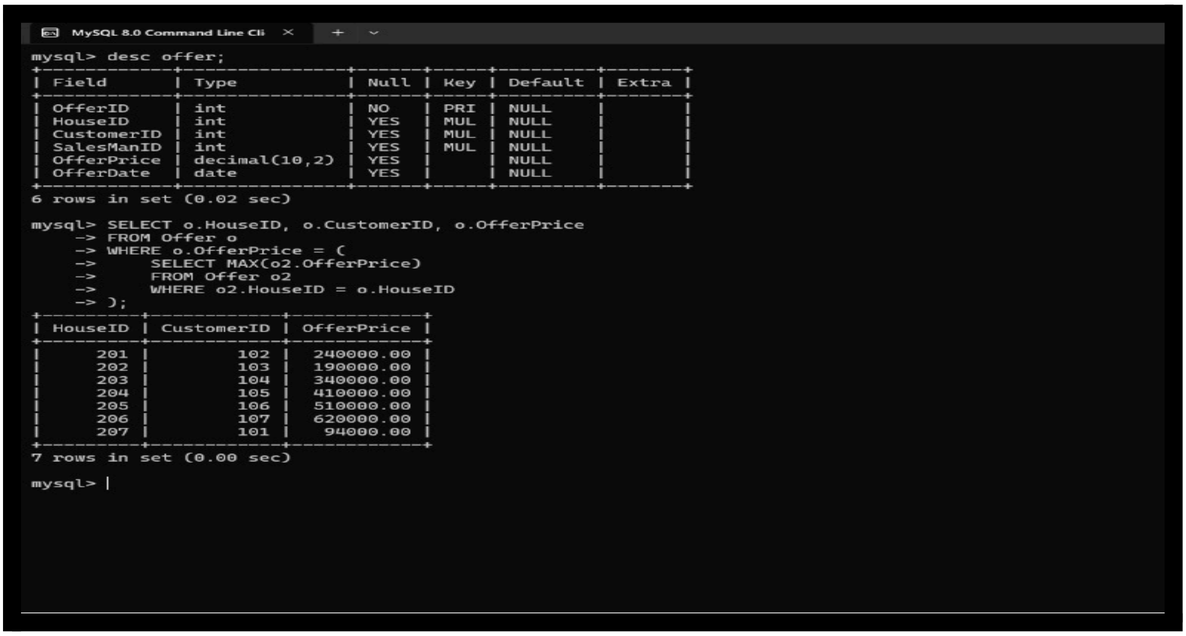
Purpose: To assess each salesman's performance by counting the number of houses sold and the total sales amount.

Fig: Query

```
mysql> SELECT
    ->     s.Name AS SalesManName,
    ->     COUNT(sa.SaleID) AS HousesSold,
    ->     SUM(sa.SalePrice) AS TotalSales
    -> FROM
    ->     SalesMan s
    -> LEFT JOIN
    ->     Sale sa ON s.SalesManID = sa.SalesManID
    -> GROUP BY
    ->     s.Name
    -> ORDER BY
```

2) **Retrieve Highest Offer for Each House**

Purpose: To determine the highest offer made on each property.



```
mysql> desc offer;
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| OfferID     | int           | NO   | PRI | NULL    |       |
| HouseID     | int           | YES  | MUL | NULL    |       |
| CustomerID  | int           | YES  | MUL | NULL    |       |
| SalesManID  | int           | YES  | MUL | NULL    |       |
| OfferPrice  | decimal(10,2) | YES  |     | NULL    |       |
| OfferDate   | date          | YES  |     | NULL    |       |
+-------------+---------------+------+-----+---------+-------+
6 rows in set (0.02 sec)

mysql> SELECT o.HouseID, o.CustomerID, o.OfferPrice
    -> FROM Offer o
    -> WHERE o.OfferPrice = (
    ->     SELECT MAX(o2.OfferPrice)
    ->     FROM Offer o2
    ->     WHERE o2.HouseID = o.HouseID
    -> );
+---------+------------+------------+
| HouseID | CustomerID | OfferPrice |
+---------+------------+------------+
|     201 |        102 | 240000.00  |
|     202 |        103 | 190000.00  |
|     203 |        104 | 340000.00  |
|     204 |        105 | 410000.00  |
|     205 |        106 | 510000.00  |
|     206 |        107 | 620000.00  |
|     207 |        101 |  94000.00  |
+---------+------------+------------+
7 rows in set (0.00 sec)

mysql> |
```

Fig: Query

6.4. PL/SQL Functions and Procedures

Examples:

This procedure will remove offers that have expired based on a certain expiration date. We assume there's an OfferDate column in the Offer table and a predefined number of days after which an offer is considered expired.

```
SQL> CREATE OR REPLACE PROCEDURE remove_expired_offers(p_days_threshold IN NUMBER) IS
BEGIN
  -- Delete offers that are older than the threshold in days
  DELETE FROM Offer
  WHERE OfferDate < SYSDATE - p_days_threshold;

  -- Optionally, you can output the number of rows deleted using SQL%ROWCOUNT
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' expired offers removed.');
END;
/

Procedure created.

SQL> BEGIN
  remove_expired_offers(30);
END;
/
0 expired offers removed.

PL/SQL procedure successfully completed.
```

Fig: Procedure 1

2) The purpose of the **auto_approve_offers** procedure is to automate the approval of real estate offers based on certain predefined criteria. In this case, the criterion is whether the **offer price** (OfferPrice) for a property is greater than or equal to the **square footage** (SqrFt) of the house listed in the House table.

```
SQL> CREATE OR REPLACE PROCEDURE auto_approve_offers IS
BEGIN
  -- Loop through offers where the offer price is greater than or equal to the square footage (SqrFt)
  FOR rec IN (SELECT o.OfferID, o.OfferPrice, h.SqrFt
              FROM Offer o
              JOIN House h ON o.HouseID = h.HouseID
              WHERE o.OfferPrice >= h.SqrFt)
  LOOP
    -- Update the offer status to 'Approved'
    UPDATE Offer
    SET Status = 'Approved'
    WHERE OfferID = rec.OfferID;
  END LOOP;
END;
/

Procedure created.

SQL> BEGIN
  auto_approve_offers;
END;
/

PL/SQL procedure successfully completed.
```

Fig: Procedure 2

3)Th **2)** func fic city. The function is designed to provide insights into the real estate market by giving the average sale price for properties in a given city.

```
SQL> CREATE OR REPLACE FUNCTION get_avg_sale_price(city_name IN VARCHAR2)
RETURN NUMBER IS
  avg_price NUMBER;
BEGIN
  -- Corrected the column name from SaleAmount to SalePrice
  SELECT AVG(sa.SalePrice)
  INTO avg_price
  FROM Sale sa
  JOIN House h ON sa.HouseID = h.HouseID
  WHERE h.City = city_name;

  RETURN avg_price;
END;
```

Fig: Procedure 3

4)The purpose of the **get_house_count_by_city(city_name IN VARCHAR2)** function is to return the **total number of houses** available in a specified city. This function helps in analyzing how many properties are listed or available in a particular city based on the House table.

```
SQL> CREATE OR REPLACE FUNCTION get_house_count_by_city(city_name IN VARCHAR2)
     RETURN NUMBER IS
       house_count NUMBER;
     BEGIN
       -- Count the number of houses available in a specific city
       SELECT COUNT(*)
       INTO house_count
       FROM House
       WHERE City = city_name;

       RETURN house_count;
     END;
     /

Function created.

SQL> SELECT get_house_count_by_city('New York') AS house_count FROM dual;

HOUSE_COUNT
-----------
          0
```

Fig: Procedure 4

6.5. Triggers

Examples: The **purpose of the trigger prevent_duplicate_sale** is to ensure **data integrity** by preventing a house from being sold more than once unless explicitly allowed. In a real estate database, once a house is sold, it should not be sold again unless the property is legally transferred back to the market. The trigger helps enforce this rule at the database level, ensuring that no duplicate sales are recorded for the same house.

```
MySQL 8.0 Command Line Cli
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use realestate;
Database changed
mysql> DELIMITER $$
mysql>
mysql> CREATE TRIGGER prevent_duplicate_sale
    -> BEFORE INSERT ON Sale
    -> FOR EACH ROW
    -> BEGIN
    ->     DECLARE house_count INT;
    ->
    ->     -- Check if the house is already sold
    ->     SELECT COUNT(*)
    ->     INTO house_count
    ->     FROM Sale
    ->     WHERE HouseID = NEW.HouseID;
    ->
    ->     -- If the house is already sold, raise an error
    ->     IF house_count > 0 THEN
    ->         SIGNAL SQLSTATE '45000'
    ->         SET MESSAGE_TEXT = 'House has already been sold.';
    ->     END IF;
    -> END $$
ERROR 1359 (HY000): Trigger already exists
mysql>
mysql> DELIMITER ;
mysql>
```

Fig: Trigger

6.6. Views

Examples:

1. **View to Show Current Ownership Status of All Houses**

Purpose: To provide a consolidated view of house ownership.

```
MySQL 8.0 Command Line Cli
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use realestate
Database changed
mysql> CREATE VIEW CurrentHouseOwnership AS
    -> SELECT
    ->     h.HouseID,
    ->     h.Address,
    ->     h.Neighborhood,
    ->     h.City,
    ->     h.Owned,
    ->     CASE
    ->         WHEN sa.BuyerID IS NOT NULL THEN c.Name
    ->         ELSE 'Not Sold'
    ->     END AS OwnerName
    -> FROM
    ->     House h
    -> LEFT JOIN
    ->     Sale sa ON h.HouseID = sa.HouseID
    -> LEFT JOIN
    ->     Customers c ON sa.BuyerID = c.CustomerID;
Query OK, 0 rows affected (0.07 sec)
```

```
mysql> select * from currenthouseownership;
+---------+--------------+--------------+------------+-------+--------------------+
| HouseID | Address      | Neighborhood | City       | Owned | OwnerName          |
+---------+--------------+--------------+------------+-------+--------------------+
|     201 | 123 Main St  | Downtown     | Metropolis |     1 | Jane Smith         |
|     202 | 456 Oak St   | Uptown       | Metropolis |     0 | Michael Johnson    |
|     203 | 789 Pine St  | Suburb       | Metropolis |     1 | Emily Davis        |
|     204 | 101 Maple St | Green Valley | Metropolis |     1 | Christopher Moore  |
|     205 | 202 Cedar St | Northside    | Metropolis |     0 | Olivia Brown       |
|     206 | 303 Birch St | East End     | Metropolis |     1 | Liam Wilson        |
|     207 | 404 Elm St   | West End     | Metropolis |     0 | John Doe           |
|     208 | 789 Cedar St | Northside    | Metropolis |     1 | Jane Smith         |
+---------+--------------+--------------+------------+-------+--------------------+
8 rows in set (0.02 sec)
```

Fig: View

2. **View to display performance across salesperson**

Purpose: To provide quick review of all salesperson.

```
mysql> CREATE VIEW SalesPerformanceView AS
    -> SELECT
    ->     s.Name AS Salesperson,  -- Changed SalesManName to Name as per the table structure
    ->     s.BirthDate,
    ->     s.Email,
    ->     s.PhoneNum,
    ->     h.HouseID,
    ->     h.Address,
    ->     c_buyer.Name AS Buyer,  -- Buyer information
    ->     sa.SalePrice,
    ->     sa.SaleDate
    -> FROM Sale sa
    -> JOIN SalesMan s ON sa.SalesManID = s.SalesManID
    -> JOIN House h ON sa.HouseID = h.HouseID
    -> JOIN Customers c_buyer ON sa.BuyerID = c_buyer.CustomerID;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> select * from SalesPerformanceView;
+---------------+------------+---------------------------+------------+---------+---------------+-------------------+-----------+------------+
| Salesperson   | BirthDate  | Email                     | PhoneNum   | HouseID | Address       | Buyer             | SalePrice | SaleDate   |
+---------------+------------+---------------------------+------------+---------+---------------+-------------------+-----------+------------+
| Alice Johnson | 1985-05-15 | alice.johnson@gmail.com   | 9871234560 |     201 | 123 Main St   | Jane Smith        | 250000.00 | 2024-01-15 |
| Bob Williams  | 1990-08-22 | bob.williams@gmail.com    | 8765123490 |     202 | 456 Oak St    | Michael Johnson   | 200000.00 | 2024-02-01 |
| Alice Johnson | 1985-05-15 | alice.johnson@gmail.com   | 9871234560 |     203 | 789 Pine St   | Emily Davis       | 350000.00 | 2024-03-10 |
| Bob Williams  | 1990-08-22 | bob.williams@gmail.com    | 8765123490 |     204 | 101 Maple St  | Christopher Moore | 400000.00 | 2024-04-22 |
| Bob Williams  | 1990-08-22 | bob.williams@gmail.com    | 8765123490 |     205 | 202 Cedar St  | Olivia Brown      | 500000.00 | 2024-05-30 |
| Alice Johnson | 1985-05-15 | alice.johnson@gmail.com   | 9871234560 |     206 | 303 Birch St  | Liam Wilson       | 600000.00 | 2024-06-15 |
| Bob Williams  | 1990-08-22 | bob.williams@gmail.com    | 8765123490 |     207 | 404 Elm St    | John Doe          |  90000.00 | 2024-07-01 |
| Alice Johnson | 1985-05-15 | alice.johnson@gmail.com   | 9871234560 |     208 | 789 Cedar St  | Jane Smith        | 275000.00 | 2024-03-10 |
+---------------+------------+---------------------------+------------+---------+---------------+-------------------+-----------+------------+
8 rows in set (0.06 sec)
```

Fig: View

7. Conclusion

The **Real Estate Database Management System** effectively addresses the core requirements of managing real estate operations by leveraging a well-designed relational database. The implementation of primary and foreign keys ensures data integrity, while advanced SQL queries facilitate comprehensive data retrieval and reporting. PL/SQL functions and triggers automate critical business processes, enhancing operational efficiency and reducing manual intervention.

Overall, the system offers a scalable and robust solution for real estate data management, with the potential for future enhancements such as integrating with web applications, implementing user authentication, and expanding reporting capabilities.