DATABASE MANAGEMENT SYSTEM (CS2207) LABORATORY MANUAL & RECORD

B.Tech (Common for CSE II YEARS)
(With effect from 2022-23 admitted batches)
(II YEAR- II SEM)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SANKETIKA VIDYA PARISHAD ENGINEERING COLLEGE
(APPROVED BY AICTE, AFFILIARED TO ANDHRA UNIVERSITY, ACCREDITED BY NAAC-A GRADE, ISO 9001:2015 CERFIFIED)
PM PALEM, VISAKHAPATNAM-41,
www.svpce.edu.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE VISION AND MISSION

VISION

To be a premier institute of knowledge of share quality research and development technologies towards national buildings

MISSION

1. Develop the state of the art environment of high quality of learning
2. Collaborate with industries and academic towards training research innovation and entrepreneurship
3. Create a platform of active participation co-circular and extra- circular activities

DEPARTMENT VISION AND MISSION

VISION

To impart quality education for producing highly talented globally recognizable techno carts and entrepreneurs with innovative ideas in computer science and engineering to meet industrial needs and societal expectations

MISSION

- To impart high standard value-based technical education in all aspects of Computer Science and Engineering through the state of the art infrastructure and innovative approach.
- To produce ethical, motivated, and skilled engineers through theoretical knowledge and practical applications.
- To impart the ability for tackling simple to complex problems individually as well as in a team.
- To develop globally competent engineers with strong foundations, capable of “out of the box” thinking so as to adapt to the rapidly changing scenarios requiring socially conscious green computing solutions.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

Graduates of B. Tech in computer science and Engineering Programme shall be able to

PEO1: Strong foundation of knowledge and skills in the field of Computer Science and Engineering.

PEO2: Provide solutions to challenging problems in their profession by applying computer engineering theory and practices.

PEO3: Produce leadership and are effective in multidisciplinary environment.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO1: Ability to design and develop computer programs and understand the structure and development methodologies of software systems.

PSO2: Ability to apply their skills in the field of networking, web design, cloud computing and data analytics.

PSO3: Ability to understand the basic and advanced computing technologies towards getting employed or to become an entrepreneur

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOMES

1. Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem Analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

6. The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, give and receive clear instructions.

11. Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-Long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE OBJECTIVES

- To introduce to a commercial DBMS such as ORACLE.
- To learn and practice SQL commands for schema creation, data manipulation.
- To learn conceptual and physical database design based on a case study.
- To apply database design stages by studying a case study.

COURSE OUTCOMES (CO)

After completion of the course, students will be able to:

- CO1: The student is exposed to a commercial RDBMS environment such as ORACLE.
- CO2:The student will learn SQL commands for data definition and manipulation.
- CO3:The student understands conceptual through physical data base design.
- CO4:The student takes up a case study and applies the design steps
- CO5:Demonstrate an understanding of fundamental database concepts, such as data models, database design, and schema development.
- CO6:Write, execute, and debug SQL queries for data retrieval, manipulation, and management in a relational data

COMPUTER SCIENCE AND ENGINEERING
GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.

2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.

3. Student should enter into the laboratory with:

a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.

b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.

c. Proper Dress code and Identity card.

4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.

5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.

6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.

7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.

8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, mis behaviors with the staff and systems etc., will attract severe punishment.

9. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.

10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly

CS2207 DATABASE MANAGEMENT SYSTEMS LAB

Syllabus

Features of a commercial RDBMS package such as ORACLE/DB2, MS Access, MYSQL & Structured Query Language (SQL) used with the RDBMS.

I. Laboratory Exercises Should Include:

a. Defining Schemas for Applications,
b. Creation of Database,
c. Writing SQL Queries,
d. Retrieve Information from Database,
e. Creating Views
f. Creating Triggers
g. Normalization up to Third Normal Form
h. Use of Host Languages,
i. Interface with Embedded SQL,
j. Use of Forms
k. Report Writing

II. Some sample applications are given below:

1. Accounting Package for Shops,
2. Database Manager for Magazine Agency or Newspaper Agency,
3. Ticket Booking for Performances,
4. Preparing Greeting Cards & Birthday Cards
5. Personal Accounts - Insurance, Loans, Mortgage Payments, Etc.,
6. Doctor's Diary & Billing System
7. Personal Bank Account
8. Class Marks Management
9. Hostel Accounting
10. Video Tape Library,
11. History of Cricket Scores,
12. Cable TV Transmission Program Manager,
13. Personal Library.
14. Sailors Database
15. Suppliers and Parts Database

TABLE OF CONTENTS

I.LABORATORY EXERCISES SHOULD INCLUDE:

A. DEFINING SCHEMAS FOR APPLICATIONS

Steps to Define Schemas for Applications:

1. Understand Application Requirements
    - Identify the purpose of the application.
    - Understand user needs, workflows, and the types of data that will be managed.
    - Gather requirements for functionality, scalability, and performance.
2. Identify Entities and Relationships
    - Break down the data into logical entities or objects (e.g., User, Order, Product).
    - Determine the relationships between these entities (e.g., one-to-many, many-to-many).
3. Choose a Database Type
    - Relational Database: Use structured schemas with predefined tables (e.g., SQL databases like MySQL, PostgreSQL).
    - NoSQL Database: Use flexible, schema-less designs for unstructured or semi-structured data (e.g., MongoDB, Cassandra).
4. Define Attributes for Each Entity
    - Specify fields for each entity, such as name, type, and constraints. Example

```
CREATE TABLE Users (
   UserID INT PRIMARY KEY,
   UserName VARCHAR(50) NOT NULL,
   Email VARCHAR(100) UNIQUE NOT NULL
);
```

5. Establish Constraints
    - Primary Key: Unique identifier for each record.
    - Foreign Key: Link between related tables.
    - Unique Constraint: Ensure no duplicate values in a column.
    - Check Constraints: Define specific rules for data (e.g., age > 18).
6. Normalize Data (for Relational Databases)
    - Reduce redundancy by dividing data into smaller tables and linking them through relationships.
    - Example: Separate a "Users" table from an "Orders" table.
7. Define Indexes
    - Improve query performance by creating indexes on frequently searched fields.
8. Implement Schema Validation (NoSQL Databases)
    - Use JSON Schema or other validation tools to enforce data structure in NoSQL databases. Example:

```
{
  "bsonType": "object",
  "required": ["name", "email"],
  "properties": {
    "name": {
      "bsonType": "string",
      "description": "must be a string and is required"
    },
    "email": {
      "bsonType": "string",
      "description": "must be a string and is required"
    }
  }}
```

9. Consider Scalability

- Use partitioning or sharding for large datasets.
- Optimize the schema for read-heavy or write-heavy operations.

10. Test the Schema

- Create sample data and test use cases.
- Ensure schema meets performance and functional requirements.

11. Maintain and Evolve the Schema

- Regularly update the schema as application requirements change.
- Use migration tools (e.g., Flyway, Liquibase) to manage schema changes in relational databases.

B.CREATING OF DATABASE

Steps to Create a Database

1. Select a Database Management System (DBMS)

- Choose a DBMS based on your application's requirements.
  - Relational Databases: MySQL, PostgreSQL, Oracle, SQL Server.
  - NoSQL Databases: MongoDB, Cassandra, Firebase, DynamoDB.

2. Plan the Database Structure

- Identify the data entities and their relationships.
- Decide on the schema design (normalized for relational databases, flexible for NoSQL).

3. Set Up the Database Environment

- Install the DBMS on your server or use a cloud-based solution like AWS RDS or Google Cloud Firestore.
- Configure access settings (e.g., admin accounts, security rules).

4. Create the Database

- Use a DBMS interface (command-line tools, GUIs, or management portals) to create the database.

Example for Relational Database:

```
CREATE DATABASE MyApplicationDB;
```

Example for NoSQL Database (MongoDB):

```
use MyApplicationDB;
```

5. Define Tables or Collections

- **Relational Databases**: Create tables with columns, data types, and constraints.

Sql:

```
CREATE TABLE Users (
UserID INT PRIMARY KEY,
UserName VARCHAR(50) NOT NULL,
Email VARCHAR(100) UNIQUE NOT NULL,
CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

NoSQL Databases: Define collections and insert documents.

Javascript:

```
db.createCollection("Users");
db.Users.insert({
   UserID: 1,
   UserName: "JohnDoe",
   Email: "johndoe@example.com"
});
```

6. Define Relationships (if applicable)

- Use **foreign keys** in relational databases to connect tables.

```
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   UserID INT,
   OrderDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

7. Set Up Indexes

- Create indexes to optimize queries and improve performance
  CREATE INDEX idx_email ON Users (Email)

NoSQL Example: MongoDB automatically indexes _id, but additional indexes can be added.

```
db.Users.createIndex({ Email: 1 });
```

8. Insert Initial Data

- Populate the database with initial or sample data.

SQL Example:

```
INSERT INTO Users (UserID, UserName, Email)
VALUES (1, 'JohnDoe', 'johndoe@example.com');
```

NoSQL Example:

```
db.Users.insertMany([
   { UserID: 1, UserName: "JohnDoe", Email: "johndoe@example.com" },
   { UserID: 2, UserName: "JaneSmith", Email: "janesmith@example.com" }
]);
```

9. Configure Access Permissions

- Grant roles and privileges to users

```
GRANT ALL PRIVILEGES ON MyApplicationDB.* TO 'app_user'@'localhost' IDENTIFIED BY 'password
FLUSH PRIVILEGES;
```

10. Test the Database

- Run queries to ensure the structure and data meet requirements

```
SELECT * FROM Users;
```

11. Back Up the Database

- Set up automated backups to prevent data loss.
- Example:

```
mysqldump -u root -p MyApplicationDB > backup.sql
```

12. Monitor and Maintain

- Use monitoring tools to track database performance and security.
- Update the schema or configuration as needed for scalability or new features.

C.WRITING SQL QUERIES

DML COMMANDS :

Data Manipulation Language (DML) commands in SQL are used for managing data within schema objects. These commands are responsible for inserting, updating, deleting, and retrieving data from the database tables. The primary DML commands are INSERT, UPDATE, DELETE, and SELECT

- Insert
- Update
- Delete
- Select

Table Structure: employees
Let's consider a table employees with the following structure: Sql code :

```
CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   first_name VARCHAR(50), last_name
   VARCHAR(50),
   email VARCHAR(100),
   hire_date DATE, salary
   DECIMAL(10, 2)
);
```

1. INSERT Command

The INSERT command is used to add new rows to a table. Example:

Sql code :

```
INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, salary)
VALUES (1, 'John', 'Doe', 'john.doe@example.com', '2023-01-15', 50000.00);
```

2. UPDATE Command

The UPDATE command is used to modify existing rows in a table.

Sql code :

Example:

```
UPDATE employees SET
salary = 55000.00
WHERE employee_id = 1;
```

3. DELETE Command

The DELETE command is used to remove rows from a table. Example:

Sql code :

```
DELETE FROM employees WHERE
employee_id = 1;
```

4. SELECT Command

The SELECT command is used to retrieve data from a table. Example:

Sql code :

```
SELECT employee_id, first_name, last_name, email, hire_date, salary FROM employees;
```

1. Insert multiple records.
2. Update one of the records.

3. Retrieve the records.
4. Delete a record.

Step 1: Insert multiple records

Sql code :

```
INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, salary)
VALUES
(1, 'John', 'Doe', 'john.doe@example.com', '2023-01-15', 50000.00),
(2, 'Jane', 'Smith', 'jane.smith@example.com', '2023-02-20', 60000.00),
(3, 'Robert', 'Brown', 'robert.brown@example.com', '2023-03-10', 70000.00); Step 2:
```

Update one of the records

Sql code :

```
UPDATE employees SET
salary = 65000.00
WHERE employee_id = 2; Step
```

3: Retrieve the records

Sql code :

```
SELECT employee_id, first_name, last_name, email, hire_date, salary FROM employees;
```

Step 4: Delete a record

Sql code :

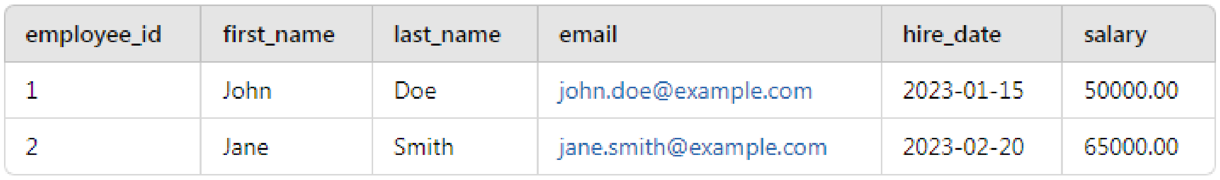
```
DELETE FROM employees WHERE
employee_id = 3;
```

Final State of the Table

After performing the above steps, the employees table would have the following records:

SELECT employee_id, first_name, last_name, email, hire_date, salary FROM employees;

Result:



employee_id	first_name	last_name	email	hire_date	salary
1	John	Doe	john.doe@example.com	2023-01-15	50000.00
2	Jane	Smith	jane.smith@example.com	2023-02-20	65000.00

D.RETRIEVE INFORMATION FROM DATABASE

1. Retrieving Information from a Relational Database (SQL)

SQL is used for querying data in relational databases like MySQL, PostgreSQL, or SQL Server. Common SQL commands include SELECT, JOIN, WHERE, ORDER BY, etc.

Basic Retrieval: SELECT Query

The SELECT statement retrieves data from a table.

```
SELECT * FROM Users;
```

This retrieves all columns for all rows in the Users table.

Retrieving Specific Columns:

To fetch specific columns:

```
SELECT UserName, Email FROM Users;
```

Filtering Data: WHERE Clause

Use the WHERE clause to filter results.

```
SELECT UserName, Email FROM Users WHERE UserID = 1;
```

Sorting Data: ORDER BY Clause

The ORDER BY clause allows you to sort the results.

```
SELECT UserName, Email FROM Users ORDER BY UserName ASC;
```

Combining Multiple Tables: JOIN Clause

To retrieve data from multiple tables, use JOIN.

```
SELECT Users.UserName, Orders.OrderID, Orders.OrderDate

FROM Users

JOIN Orders ON Users.UserID = Orders.UserID;
```

This query retrieves UserName from the Users table and OrderID, OrderDate from the Orders table where the UserID matches.

```
SELECT UserID, COUNT(OrderID) AS OrderCount

FROM Orders

GROUP BY UserID;
```

Limiting Results: LIMIT Clause

Use the LIMIT clause to limit the number of rows returned.

```
SELECT * FROM Users LIMIT 10;
```

E. CREATING VIEWS PROGRAM

Creating views in SQL allows you to simplify complex queries, encapsulate data logic, and present data in a specific format without altering the underlying tables. A view is essentially a virtual table based on a SELECT query.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

Table Structure: Studentsand Courses

Sql code :

```
CREATE TABLE students ( student_id
    INT PRIMARY KEY, first_name
    VARCHAR(50), last_name
    VARCHAR(50), enrollment_year INT
);


CREATE TABLE courses ( course_id INT
    PRIMARY KEY, course_name
    VARCHAR(100), instructor
    VARCHAR(50)
);
```

1. Simple View

Create a simple view that combines first name and last name of students into a single column.

View: student_names

Sql code :

```
CREATE VIEW student_names AS
SELECT student_id, CONCAT(first_name, ' ', last_name) AS full_name FROM students;
```

Usage:

Sql code :

```
SELECT * FROM student_names;
```

2. View with Join

Create a view that shows a list of students along with the courses they are enrolled in.

Table: enrollments

Sql code :

```
CREATE TABLE enrollments ( enrollment_id
    INT PRIMARY KEY, student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id), FOREIGN
    KEY (course_id) REFERENCES courses(course_id)
);
```

View: student_course_enrollments

Sql code :

```
CREATE VIEW student_course_enrollments AS
SELECT s.student_id, CONCAT(s.first_name, ' ', s.last_name) AS full_name, c.course_name, c.instructor
FROM students s
JOIN enrollments e ON s.student_id = e.student_id JOIN
courses c ON e.course_id = c.course_id;
```

Usage:

Sql code :

```
SELECT * FROM student_course_enrollments;
```

3. Aggregated View

Create a view that shows the number of courses each student is enrolled in.

View: student_course_count

Sql code :

```
CREATE VIEW student_course_count AS
SELECT s.student_id, CONCAT(s.first_name, ' ', s.last_name) AS full_name, COUNT(e.course_id) AS
course_count
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id GROUP BY
s.student_id, s.first_name, s.last_name;
```

Usage:

Sql code :

```
SELECT * FROM student_course_count;
```

Step 1: Insert data into tables Inserting

values into different tables Sql code :

```
INSERT INTO students (student_id, first_name, last_name, enrollment_year) VALUES
(1, 'John', 'Doe', 2023),
(2, 'Jane', 'Smith', 2022),
(3, 'Robert', 'Brown', 2023);
```

Sql code :

```
INSERT INTO courses (course_id, course_name, instructor) VALUES (1, 'Database
Systems', 'Dr. White'),
(2, 'Operating Systems', 'Prof. Black'), (3, 'Data
Structures', 'Dr. Green');
```

Sql code :

```
INSERT INTO enrollments (enrollment_id, student_id, course_id) VALUES (1, 1, 1),
(2, 1, 2),
(3, 2, 1),
(4, 3, 3);
```

Step 2: Create and use the views

Sql code :

```
-- View: student_names

CREATE VIEW student_names AS
SELECT student_id, CONCAT(first_name, ' ', last_name) AS full_name FROM students;
```

Sql code :

```
-- View: student_course_enrollments

CREATE VIEW student_course_enrollments AS
```

```
SELECT s.student_id, CONCAT(s.first_name, ' ', s.last_name) AS full_name, c.course_name, c.instructor
FROM students s
JOIN enrollments e ON s.student_id = e.student_id JOIN
courses c ON e.course_id = c.course_id;
```

Sql code :

```
-- View: student_course_count

CREATE VIEW student_course_count AS
SELECT s.student_id, CONCAT(s.first_name, ' ', s.last_name) AS full_name, COUNT(e.course_id) AS
course_count
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id GROUP BY
s.student_id, s.first_name, s.last_name;
```

Usage Examples:

Sql code :

```
-- View: student_names
SELECT * FROM student_names;
```

Sql code :

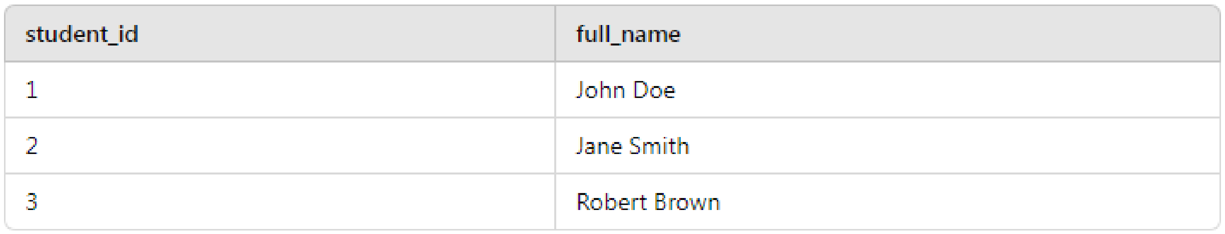
```
-- View: student_course_enrollments
SELECT * FROM student_course_enrollments;
```

Sql code :

```
-- View: student_course_count
SELECT * FROM student_course_count;
```

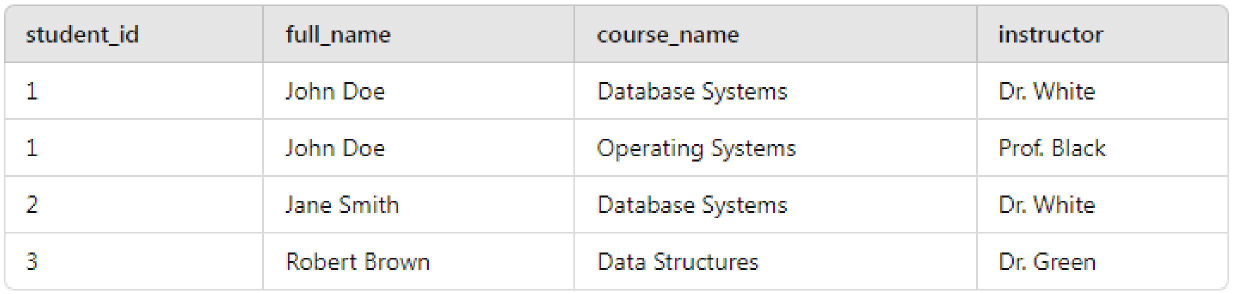
Expected Results

Result of student_names:



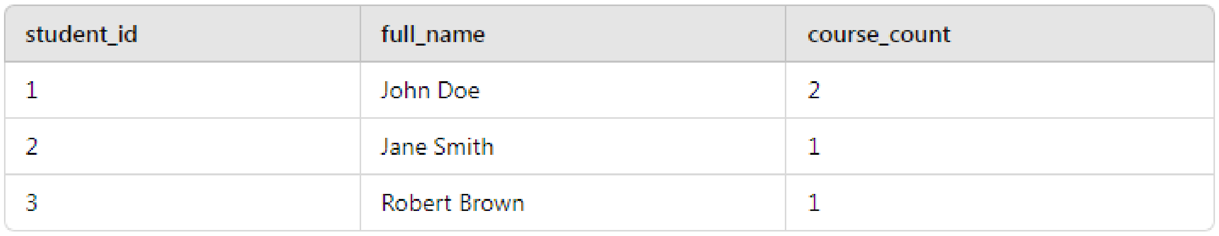
student_id	full_name
1	John Doe
2	Jane Smith
3	Robert Brown

Result of student_course_enrollments:



student_id	full_name	course_name	instructor
1	John Doe	Database Systems	Dr. White
1	John Doe	Operating Systems	Prof. Black
2	Jane Smith	Database Systems	Dr. White
3	Robert Brown	Data Structures	Dr. Green

Result of student_course_count:



student_id	full_name	course_count
1	John Doe	2
2	Jane Smith	1
3	Robert Brown	1

F. CREATING TRIGGERS PROGRAM

Creating triggers in SQL allows you to automate certain actions based on events that occur in your database tables. Triggers are typically used for enforcing business rules, validating data, and maintaining audit trails. Here, we'll go through a simple and easy example of creating triggers using a table named employees

1. **Trigger Definition**: The trigger after_employee_insert is defined to fire after an insert operation on the employees table.
2. **Trigger Action**: When a new row is inserted into the employees table, the trigger inserts a new row into the employee_audit table. The new row contains the employee_id of the inserted employee, the action 'INSERT', and the current timestamp (NOW()).

Table Structure: employees

Let's consider a table employees with the following structure: Sql code :

```
CREATE TABLE employees ( employee_id
    INT PRIMARY KEY, first_name
    VARCHAR(50), last_name
    VARCHAR(50),
    email VARCHAR(100),
    hire_date DATE,
    salary DECIMAL(10, 2)
);
```

Creating a Trigger

Let's create a simple trigger that automatically updates a log table whenever a new record is inserted into the employees table.

Step-by-Step Example

1. **Create a log table:** employee_audit
2. **Create a trigger that logs the insert operation**

Step 1: Create the log table

First, create a table to store audit logs. This table will store information about the insert operations on the employees table

Sql code:

```
CREATE TABLE employee_audit (
    audit_id INT PRIMARY KEY AUTO_INCREMENT,
    employee_id INT, action
    VARCHAR(50),
    action_time DATETIME
);
```

Step 2: Create the trigger

Next, create a trigger that fires after an insert operation on the employees table. This trigger will insert a record into the employee_audit table.

Sql code:

```
CREATE TRIGGER after_employee_insert AFTER
INSERT ON employees
FOR EACH ROW BEGIN
    INSERT INTO employee_audit (employee_id, action, action_time) VALUES
    (NEW.employee_id, 'INSERT', NOW());
END;
```

Testing the Trigger

To see the trigger in action, let's insert a new record into the employees table and check the employee_audit table.

Insert a record into the employees table

Sql code:

```
INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, salary)
VALUES (1, 'John', 'Doe', 'john.doe@example.com', '2023-01-15', 50000.00);
```

Check the employee_audit table sql

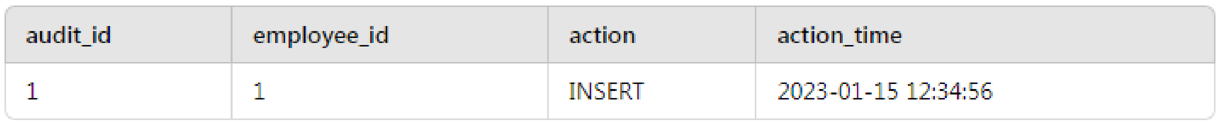
Sql code:

```
SELECT * FROM employee_audit;
```

Expected Result

After inserting a new record into the employees table, the employee_audit table should contain a new record logging the insert action.

employee_audit table



audit_id	employee_id	action	action_time
1	1	INSERT	2023-01-15 12:34:56

G.Normalization up to Third Normal Form PROGRAM

Normalization up to Third Normal Form (3NF) in DBMS
Normalization is the process of organizing the data in a database to minimize data redundancy and dependency. Here, we'll go through the normalization process up to 3NF with an example.

Initial Table:
Let's consider a table student_details with the following columns:

Column Name	Data Type
student_id	INT
student_name	VARCHAR(100)
Department	VARCHAR(100)
Course	VARCHAR(100)
Professor	VARCHAR(100)

Sample Data:

student_id	student_name	department	course	professor
1	John Smith	Computer Science	Data Structures	Dr. Johnson
2	Jane Doe	Computer Science	Algorithms	Dr. Johnson
3	Bob Johnson	Electrical Engineering	Circuit Analysis	Dr. Thompson
4	Alice Brown	Computer Science	Data Structures	Dr. Johnson

First Normal Form (1NF):
In 1NF, each column should have atomic values. Our table is already in 1NF, so no changes are needed.

Second Normal Form (2NF):
In 2NF, each non-prime attribute should depend on the entire primary key. In our table, student_id is the primary key, and department, course, and professor depend

on student_id. However, professor also depends on course, which is not part of the primary key. To achieve 2NF, we need to split the table into two tables:

Table 1: student_details

Column Name	Data Type
student_id	INT
student_name	VARCHAR(100)
Department	VARCHAR(100)

Table 2: course_details

Column Name	Data Type
Course	VARCHAR(100)
Professor	VARCHAR(100)

Sample Data:

student_details

student_id	student_name	department
1	John Smith	Computer Science
2	Jane Doe	Computer Science
3	Bob Johnson	Electrical Engineering
4	Alice Brown	Computer Science

course_details

Course	professor
Data Structures	Dr. Johnson
Algorithms	Dr. Johnson
Circuit Analysis	Dr. Thompson

Third Normal Form (3NF):

In 3NF, if a table is in 2NF, and a non-prime attribute depends on another non- prime attribute, then it should be moved to a separate table. In our
case, course depends on department, so we need to create a new table

department_courses: Table 1:

student_details

Column Name	Data Type
student_id	INT
student_name	VARCHAR(100)
department	VARCHAR(100)

Table 2: department_courses

Column Name	Data Type
department	VARCHAR(100)
course	VARCHAR(100)

Table 3: course_details

Column Name	Data Type
course	VARCHAR(100)
professor	VARCHAR(100)

Sample Data:
student_details

student_id	student_name	department
1	John Smith	Computer Science
2	Jane Doe	Computer Science
3	Bob Johnson	Electrical Engineering
4	Alice Brown	Computer Science

department_courses

department	course
Computer Science	Data Structures
Computer Science	Algorithms
Electrical Engineering	Circuit Analysis

course_details

course	professor
Data Structures	Dr. Johnson
Algorithms	Dr. Johnson
Circuit Analysis	Dr. Thompson

H. Host Languages in DBMS – Program

In DBMS (Database Management Systems), a host language is a programming language that is used to interact with a database. It is the language in which the database is accessed, manipulated, and queried. Host languages provide a way to write programs that can execute database operations, such as creating tables, inserting data, updating records, and retrieving data.

Host languages are typically used to:

1. Create database schema: Define the structure of the database, including tables, indexes, and relationships.
2. Insert, update, and delete data: Perform CRUD (Create, Read, Update, Delete) operations on the database.
3. Query the database: Execute SQL queries to retrieve specific data from the database.
4. Implement business logic: Write programs that enforce business rules, perform calculations, and make decisions based on data in the database.

Examples of Host Languages in DBMS:

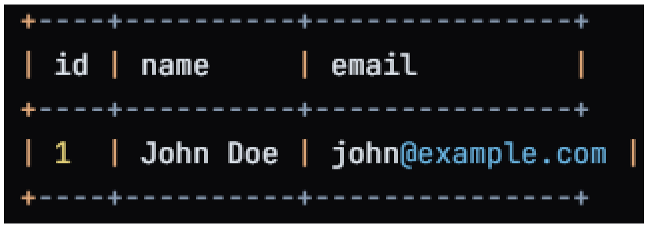
1. SQL (Structured Query Language): SQL is a standard language for managing relational databases. It is used to create, modify, and query databases.
2. Python: Python is a popular programming language that is widely used with databases. It provides libraries such as sqlite3, psycopg2, and mysql-connector-python to interact with various databases.
3. Java: Java is an object-oriented language that is commonly used with databases. It provides APIs such as JDBC (Java Database Connectivity) to interact with databases.
4. C#: C# is a modern, object-oriented language that is widely used with databases. It provides APIs such as ADO.NET to interact with databases.

Host Languages in DBMS with Outputs:

Here are some examples of host languages in DBMS with outputs:

SQL (Structured Query Language) : -

```
Create a table
CREATE TABLE customers ( id
  INT PRIMARY KEY, name
  VARCHAR(255), email
  VARCHAR(255)
);

Insert data
INSERT INTO customers (id, name, email) VALUES (1, 'John Doe', 'john@example.com');

Retrieve data
SELECT * FROM customers;
```



```
+----+----------+------------------+
| id | name     | email            |
+----+----------+------------------+
| 1  | John Doe | john@example.com |
+----+----------+------------------+
```

Python with SQLite :-

```
import sqlite3

#Connect to the database
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Create a table
cursor.execute('''
    CREATE TABLE customers (
        id INTEGER PRIMARY KEY,
        name TEXT, email
        TEXT
    )
''')

# Insert data
```

```
cursor.execute("INSERT INTO customers (id, name, email) VALUES (1, 'John Doe',
'john@example.com')")

# Retrieve data
cursor.execute("SELECT * FROM customers") rows =
cursor.fetchall()

# Print the output for
row in rows:
    print(row)

# Output:
(1, 'John Doe', 'john@example.com')
```

Java with JDBC (Java Database Connectivity) :-

```
import java.sql.*;
public class DatabaseExample {
    public static void main(String[] args) {
        // Load the JDBC driver Class.forName("com.mysql.cj.jdbc.Driver");
        // Connect to the database
        Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/example", "username",
"password");
        Statement stmt = conn.createStatement();

        // Create a table
        stmt.executeUpdate("CREATE TABLE customers (id INT PRIMARY KEY, name
VARCHAR(255), email VARCHAR(255))");

        // Insert data
        stmt.executeUpdate("INSERT INTO customers (id, name, email) VALUES (1, 'John Doe',
'john@example.com')");

        // Retrieve data
        ResultSet rs = stmt.executeQuery("SELECT * FROM customers");

        // Print the output
        while (rs.next()) {
            System.out.println(rs.getInt("id") + " " + rs.getString("name") + " " + rs.getString("email"));
        }
        // Output:
        1 John Doe john@example.com
    }
}
```

C# with ADO.NET :-

```
using System; using
System.Data;
using System.Data.SqlClient;

class DatabaseExample {
    static void Main(string[] args) {
        // Connect to the database
        string connectionString = "Server=localhost;Database=example;User
Id=username;Password=password;";
        SqlConnection conn = new SqlConnection(connectionString); conn.Open();

        // Create a table
        SqlCommand cmd = new SqlCommand("CREATE TABLE customers (id INT
PRIMARY KEY, name VARCHAR(255), email VARCHAR(255))", conn);
        cmd.ExecuteNonQuery();

        // Insert data
        cmd.CommandText = "INSERT INTO customers (id, name, email) VALUES (1, 'John
Doe', 'john@example.com')";
        cmd.ExecuteNonQuery();

        // Retrieve data
        cmd.CommandText = "SELECT * FROM customers";
        SqlDataReader reader = cmd.ExecuteReader();

        // Print the output  while
        (reader.Read()) {
            Console.WriteLine(reader["id"] + " " + reader["name"] + " " + reader["email"]);
        }

        // Output:
        1 John Doe john@example.com
    }
}
```

I.Interface with Embedded SQL – Program

An interface with embedded SQL is a programming interface that allows a program to interact with a database management system (DBMS) using SQL statements embedded in the program's code. This interface provides a way for a program to access, manipulate, and retrieve data from a database.

Embedded SQL is a technique where SQL statements are embedded directly into a program's source code, allowing the program to interact with a database without the need for a separate database interface or API. The SQL statements are typically prefixed with a special keyword, such as "EXEC SQL", to distinguish them from the rest of the program's code.

Interface with Embedded SQL in DBMS :-

In DBMS, an interface with embedded SQL provides a way for a program to interact with a database using SQL statements embedded in the program's code. This interface allows a program to:

1. Execute SQL statements to retrieve or manipulate data in a database.
2. Declare and use host variables to exchange data between the program and the database.
3. Use cursors to navigate and manipulate data in a database.

Example of Interface with Embedded SQL in DBMS with Outputs

Example 1: Embedded SQL in C Program :-

Suppose we have a C program that needs to retrieve employee data from a database table called "Employees". We can use embedded SQL to interact with the database.

C Program Code:

```
#include <stdio.h> #include
<sqlca.h>

EXEC SQL BEGIN DECLARE SECTION;
    char emp_name[20]; int
    emp_id;
EXEC SQL END DECLARE SECTION;

int main() {
    EXEC SQL CONNECT TO mydb AS conn1;
    EXEC SQL SELECT name, id INTO :emp_name, :emp_id FROM Employees WHERE id =
1;
    printf("Employee Name: %s, ID: %d\n", emp_name, emp_id); EXEC SQL
    COMMIT WORK RELEASE;
    return 0;
}
```

Output:

```
Employee Name: John Smith, ID: 1
```

Example 2: Embedded SQL in Java Program :

Suppose we have a Java program that needs to insert a new order into a database table called "Orders". We can use embedded SQL to interact with the database.

Java Program Code:

```
import java.sql.*;
public class InsertOrder {
    public static void main(String[] args) { Connection conn
        =
DriverManager.getConnection("jdbc:mydb://localhost:5432/mydb", "username", "password");
```

```
        Statement stmt = conn.createStatement();
        stmt.execute("INSERT INTO Orders (customer_id, order_date, total) VALUES (1, '2022-
01-01', 100.00)");
        System.out.println("Order inserted successfully!"); conn.close();
    }
}
```

Output:

```
Order inserted successfully!
```

Example 3: Embedded SQL in Python Program

Suppose we have a Python program that needs to retrieve product data from a database table called "Products". We can use embedded SQL to interact with the database.

Python Program Code:

```
import pyodbc

conn = pyodbc.connect("DRIVER={ODBC Driver 17 for SQL
Server};SERVER=localhost;DATABASE=mydb;UID=username;PWD=password ")
cursor = conn.cursor()
cursor.execute("SELECT * FROM Products WHERE id = 1") row =
cursor.fetchone()
print("Product Name:", row[1], "Price:", row[3])
conn.close()
```

Output:

```
Product Name: Product A Price: 10.00
```

J. Use of Forms – Program

Forms are a graphical user interface (GUI) component that allows users to interact with a system, application, or database. Forms provide a way to collect, display, and manipulate data in a structured and organized manner. Forms can be used in various contexts, including:

1. Data entry: Forms can be used to enter new data into a system or database.
2. Data editing: Forms can be used to edit existing data in a system or database.
3. Data display: Forms can be used to display data from a system or database in a user- friendly format.
4. Search and filtering: Forms can be used to search and filter data based on specific criteria.
5. Reporting: Forms can be used to generate reports based on data from a system or database.

Use of Forms in DBMS :-

In DBMS (Database Management Systems), forms are used to interact with a database, perform various tasks, and display data in a user-friendly format. Forms in DBMS provide a way to:

1. Create, edit, and delete data in a database.
2. Search and filter data based on specific criteria.
3. Generate reports based on data from a database.
4. Perform data validation and error checking.

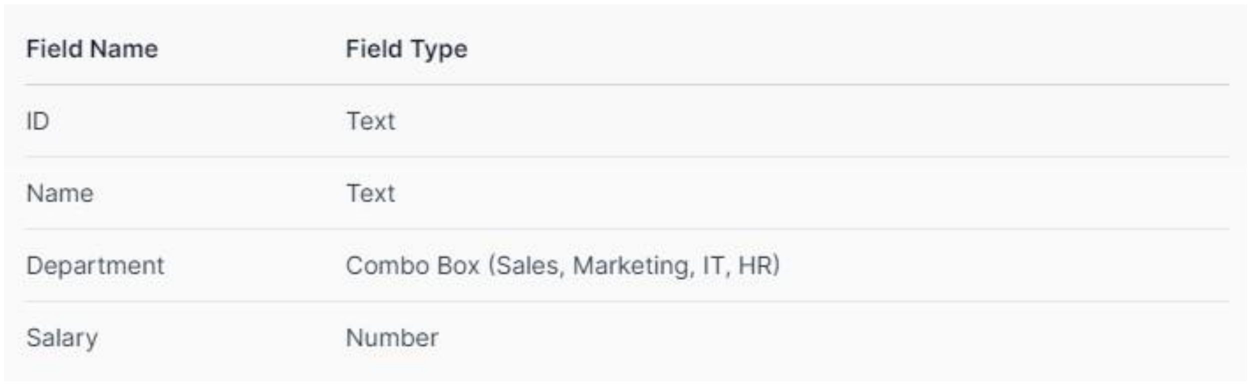
5. Improve data consistency and integrity.

Example of Use of Forms in DBMS with Outputs Example

1: Employee Data Entry Form :-

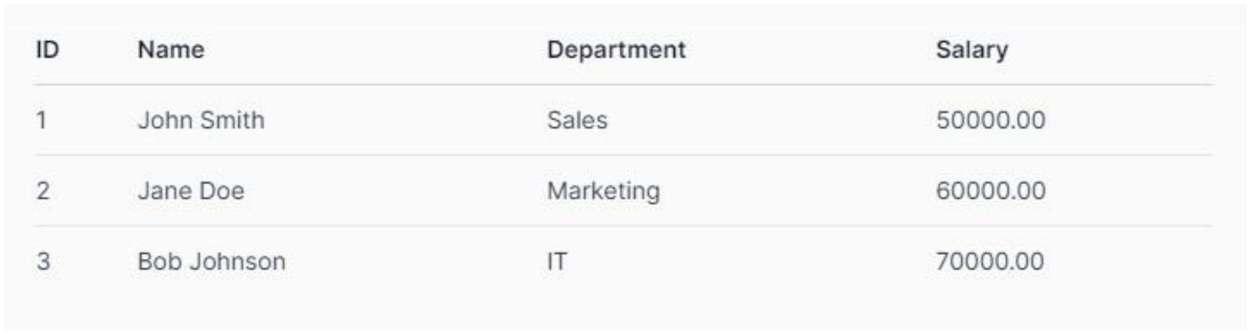
Suppose we have a database table called "Employees" with columns "ID", "Name", "Department", and "Salary". We can create a data entry form to enter new employee data into the table.

Form Design:



Field Name	Field Type
ID	Text
Name	Text
Department	Combo Box (Sales, Marketing, IT, HR)
Salary	Number

Form Output:

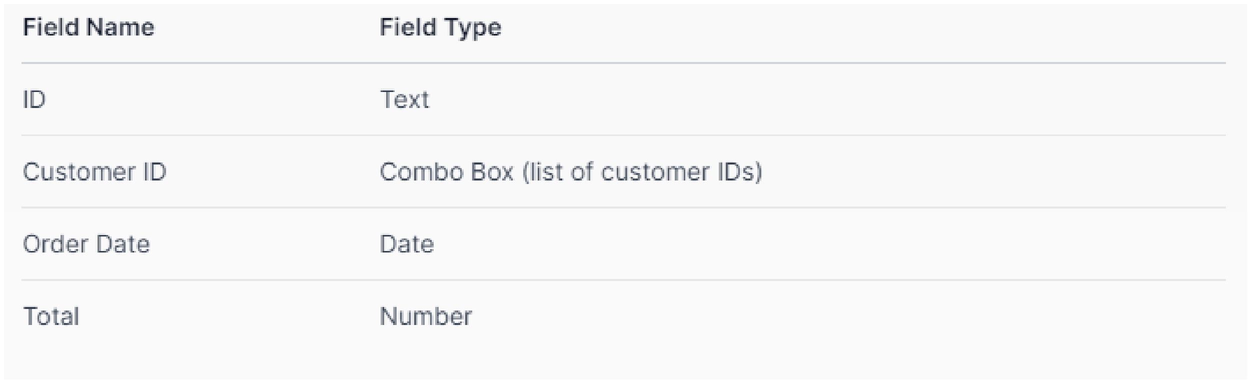


ID	Name	Department	Salary
1	John Smith	Sales	50000.00
2	Jane Doe	Marketing	60000.00
3	Bob Johnson	IT	70000.00

Example 2: Customer Order Form

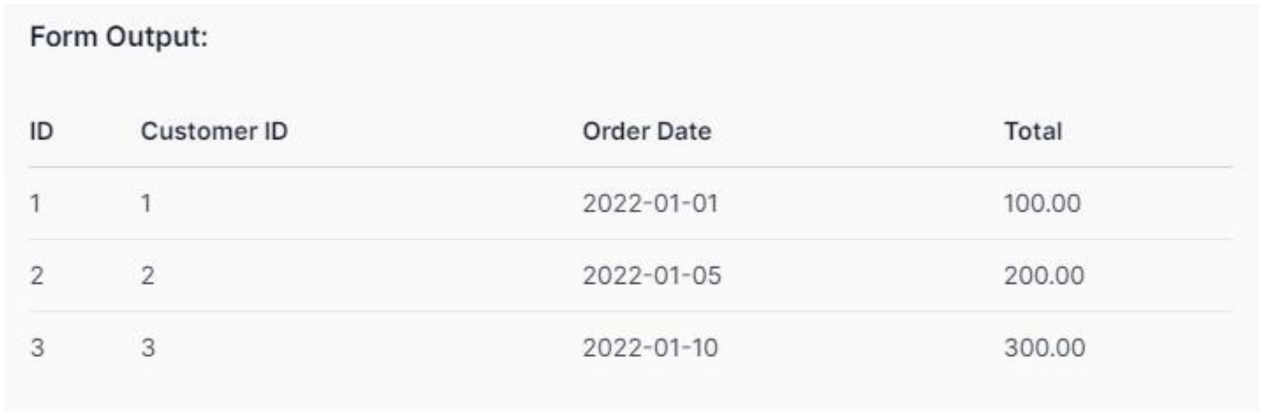
Suppose we have a database table called "Orders" with columns "ID", "Customer ID", "Order Date", and "Total". We can create a data entry form to enter new order data into the table.

Form Design:



Field Name	Field Type
ID	Text
Customer ID	Combo Box (list of customer IDs)
Order Date	Date
Total	Number

Form Output:



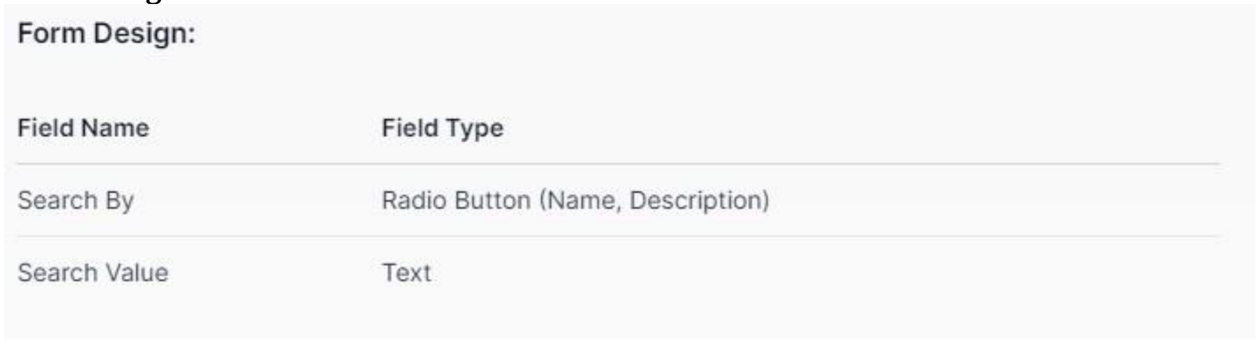
Form Output:

ID	Customer ID	Order Date	Total
1	1	2022-01-01	100.00
2	2	2022-01-05	200.00
3	3	2022-01-10	300.00

Example 3: Product Search Form

Suppose we have a database table called "Products" with columns "ID", "Name", "Description", and "Price". We can create a search form to search for products by name or description.

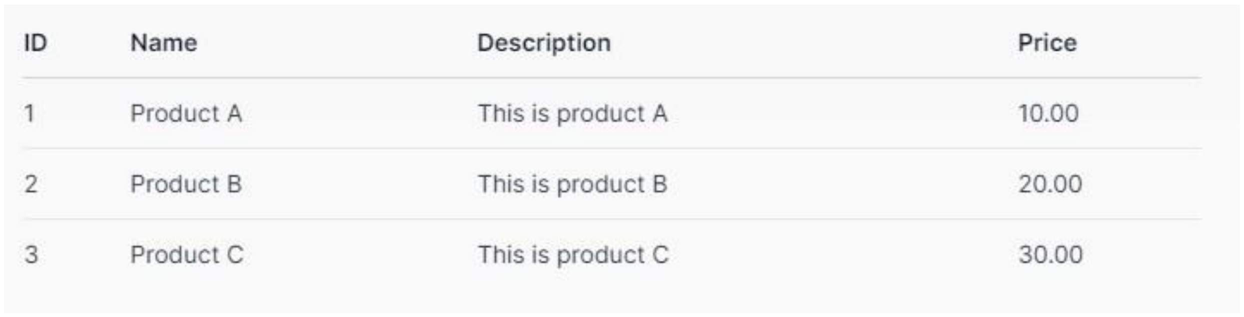
Form Design:



Form Design:

Field Name	Field Type
Search By	Radio Button (Name, Description)
Search Value	Text

Form Output:



ID	Name	Description	Price
1	Product A	This is product A	10.00
2	Product B	This is product B	20.00
3	Product C	This is product C	30.00

K. Report Writing – Program

Report writing is the process of creating a document that presents information in a clear and organized manner, often in a structured format. Reports can be used to communicate information, analyze data, and make recommendations. Report writing involves:

1. Defining the report's purpose and scope
2. Gathering and analyzing data
3. Organizing and structuring the content
4. Writing the report in a clear and concise manner
5. Editing and proofreading the report

Report Writing in DBMS :-

In DBMS (Database Management Systems), report writing is the process of creating a document that presents data from a database in a clear and organized manner. Report writing in DBMS involves:

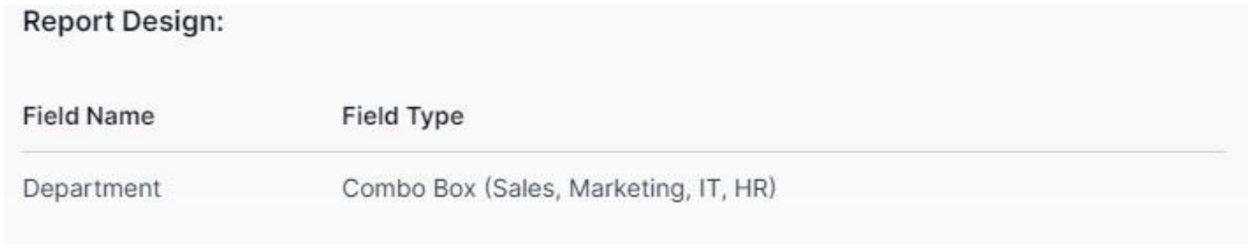
1. Defining the report's purpose and scope
2. Querying the database to retrieve the required data
3. Organizing and structuring the data
4. Writing the report in a clear and concise manner
5. Editing and proofreading the report

Example of Report Writing in DBMS with Outputs

Example 1: Employee Salary Report

Suppose we have a database table called "Employees" with columns "ID", "Name", "Department", and "Salary". We can create a report to display the salary details of all employees in a department.

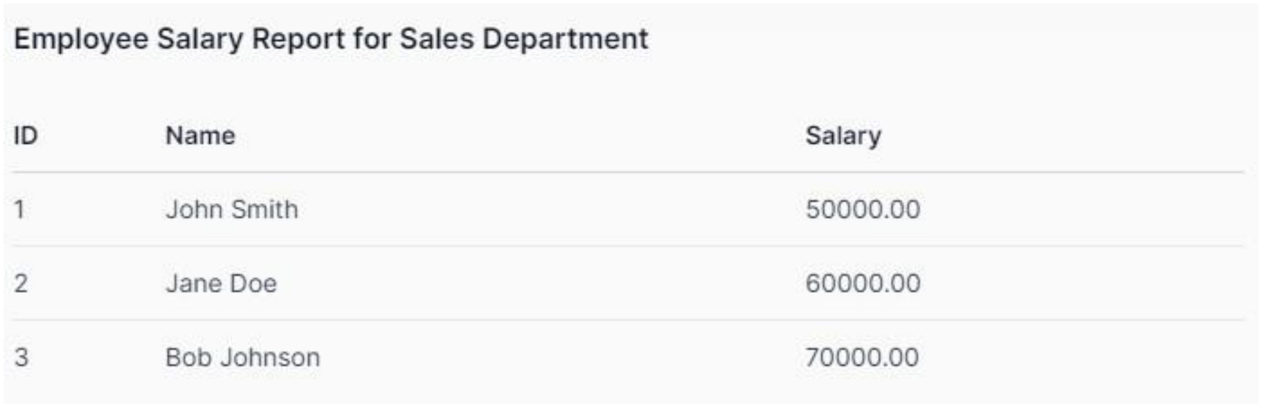
Report Design:



Report Design:

Field Name	Field Type
Department	Combo Box (Sales, Marketing, IT, HR)

Report Output :



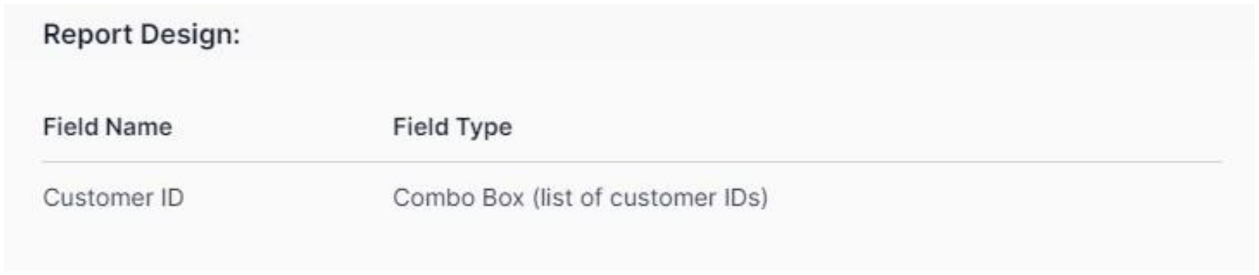
Employee Salary Report for Sales Department

ID	Name	Salary
1	John Smith	50000.00
2	Jane Doe	60000.00
3	Bob Johnson	70000.00

Example 2: Customer Order Report

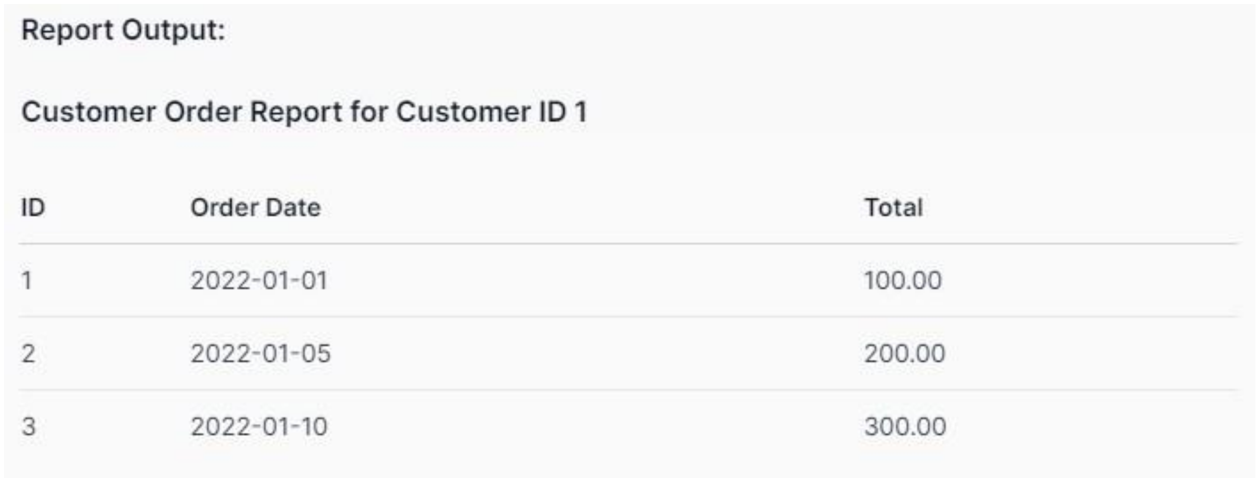
Suppose we have a database table called "Orders" with columns "ID", "Customer ID", "Order Date", and "Total". We can create a report to display the order details of all customers.

Report Design:


Report Design:

Field Name	Field Type
Customer ID	Combo Box (list of customer IDs)

Report Output :


Report Output:

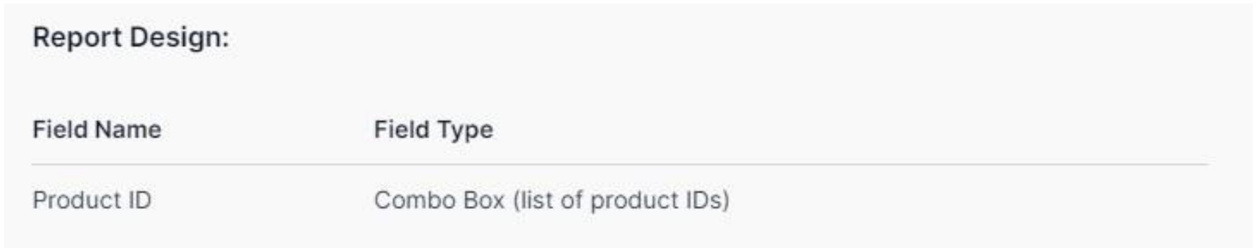
Customer Order Report for Customer ID 1

ID	Order Date	Total
1	2022-01-01	100.00
2	2022-01-05	200.00
3	2022-01-10	300.00

Example 3: Product Sales Report :

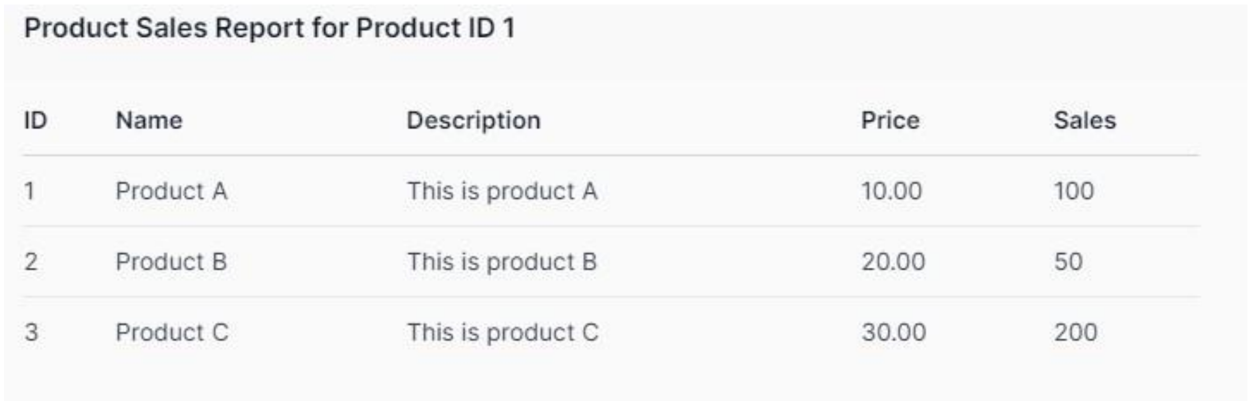
Suppose we have a database table called "Products" with columns "ID", "Name", "Description", and "Price". We can create a report to display the sales details of all products.

Report Design :


Report Design:

Field Name	Field Type
Product ID	Combo Box (list of product IDs)

Report Output :



Product Sales Report for Product ID 1

ID	Name	Description	Price	Sales
1	Product A	This is product A	10.00	100
2	Product B	This is product B	20.00	50
3	Product C	This is product C	30.00	200

II.SOME SAMPLE APPLICATIONS ARE GIVEN BELOW:

Experiment-01

Accounting package for shop

1.Shops Table

shop_id: Primary key

shop_name

shop_address

contact_person

contact_number

2.Products Table

product_id: Primary key

product_name

unit_price

stock_quantity

3.Transactions Table

transaction_id: Primary key

transaction_date

shop_id: Foreign key referencing Shops table

total_amount

4.Transaction Details Table

transaction_detail_id: Primary key

transaction_id: Foreign key referencing Transactions table

product_id: Foreign key referencing Products table

quantity

unit_price

total_price

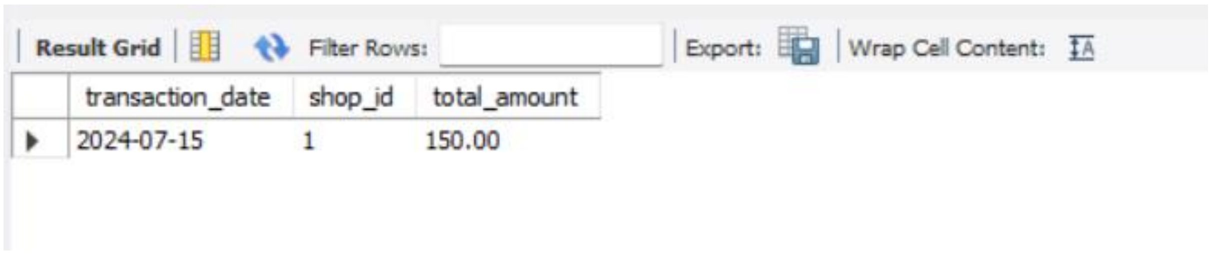
Functionality and Queries

Inserting a new transaction:

INSERT INTO Transactions (transaction_date, shop_id, total_amount)

```
VALUES ('2024-07-15', 1, 150.00);

INSERT INTO Transaction_Details (transaction_id, product_id, quantity, unit_price,
total_price)

VALUES (1, 3, 2, 50.00, 100.00),

       (1, 5, 1, 50.00, 50.00);
```

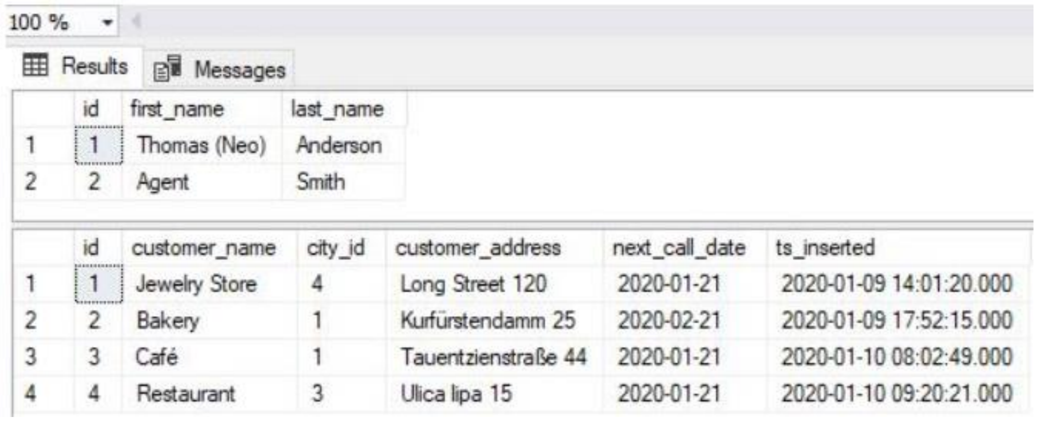
OUTPUT:



	transaction_date	shop_id	total_amount
▶	2024-07-15	1	150.00

Generating a sales report for a specific shop:

```
SELECT t.transaction_date, p.product_name, td.quantity, td.unit_price, td.total_price

FROM Transactions t

JOIN Transaction_Details td ON t.transaction_id = td.transaction_id

JOIN Products p ON td.product_id = p.product_id

WHERE t.shop_id = 1

ORDER BY t.transaction_date;
```

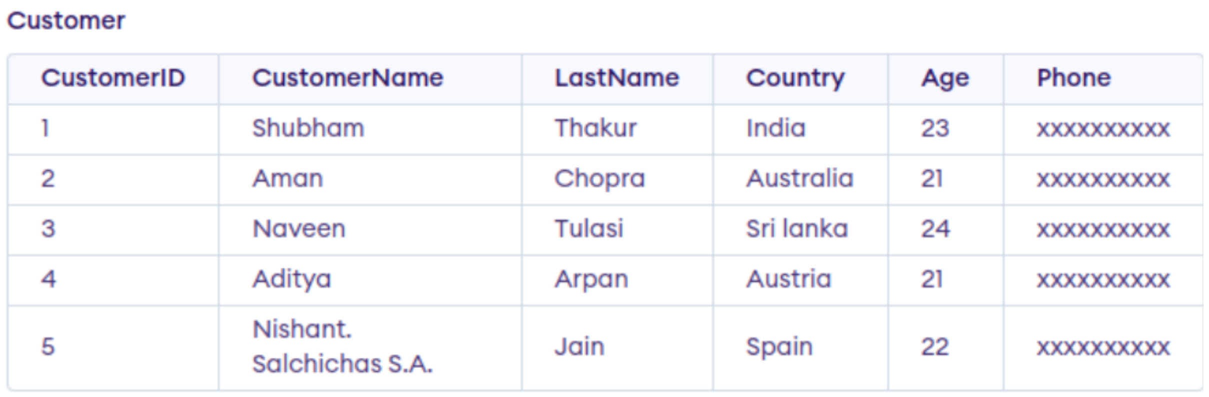


	id	first_name	last_name
1	1	Thomas (Neo)	Anderson
2	2	Agent	Smith

	id	customer_name	city_id	customer_address	next_call_date	ts_inserted
1	1	Jewelry Store	4	Long Street 120	2020-01-21	2020-01-09 14:01:20.000
2	2	Bakery	1	Kurfürstendamm 25	2020-02-21	2020-01-09 17:52:15.000
3	3	Café	1	Tauentzienstraße 44	2020-01-21	2020-01-10 08:02:49.000
4	4	Restaurant	3	Ulica lipa 15	2020-01-21	2020-01-10 09:20:21.000

Updating stock quantity after a sale:

```
 CREATE TABLE Customer(

    CustomerID INT PRIMARY KEY,

    CustomerName VARCHAR(50),

    LastName VARCHAR(50),

    Country VARCHAR(50),
   Age int(2),
  Phone int(10)
);
INSERT INTO Customer (CustomerID, CustomerName, LastName, Country, Age, Phone)
VALUES (1, 'Shubham', 'Thakur', 'India','23','xxxxxxxxxx'),
       (2, 'Aman ', 'Chopra', 'Australia','21','xxxxxxxxxx'),
       (3, 'Naveen', 'Tulasi', 'Sri lanka','24','xxxxxxxxxx'),
       (4, 'Aditya', 'Arpan', 'Austria','21','xxxxxxxxxx'),
       (5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain','22','xxxxxxxxxx');
 Select * from Customer;
```

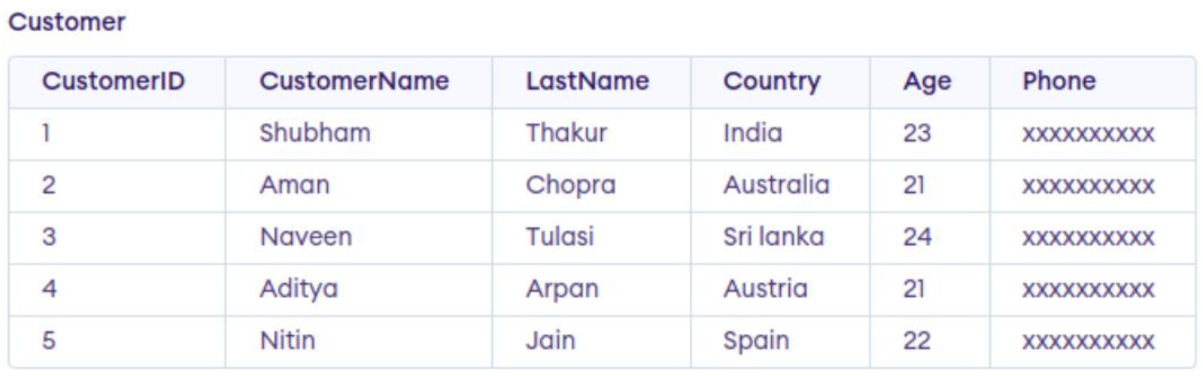


Customer

CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx

```
UPDATE Customer SET CustomerName
= 'Nitin' WHERE Age = 22;
```

OUTPUT



Customer

CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nitin	Jain	Spain	22	xxxxxxxxxx

Experiment-02

Database manager for magazine agency or newspaper agency

1.Relational Database Management System (RDBMS): RDBMS is a traditional choice for managing structured data, which is common in magazine or newspaper agencies

Some key features

Tables: A collection of related data entries, consisting of numerous columns and rows.

SQL (Structured Query Language): The standard language used to manage data in an RDBMS.

Data Integrity: RDBMSs ensure the security, accuracy, integrity, and consistency of the data stored in the database.

Relational Model: The underlying data model of an RDBMS, which organizes data into tables with related data elements.

Content Management System (CMS) Integration: Many magazine and newspaper agencies utilize CMS platforms for content creation and management.

Experiment-03

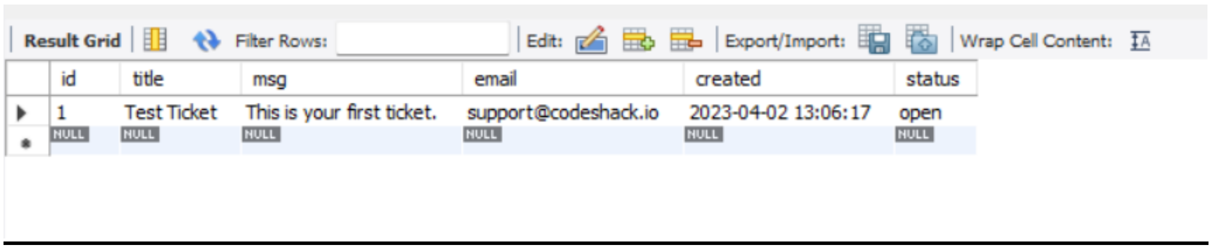
Ticket booking for performances

Designing a ticket booking system for performances in a (DBMS) involves creating a database schema that can efficiently handle the storage and retrieval of information related to events, venues, tickets, customers, bookings, and payments. Here's a simplified example schema:

```
CREATE TABLE IF NOT EXISTS `tickets` (
        `id` int(11) NOT NULL AUTO_INCREMENT,
        `title` varchar(255) NOT NULL,
        `msg` text NOT NULL,
        `email` varchar(255) NOT NULL,
        `created` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
        `status` enum('open','closed','resolved') NOT NULL DEFAULT 'open',
        PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
SELECT * FROM TICKETS;
```

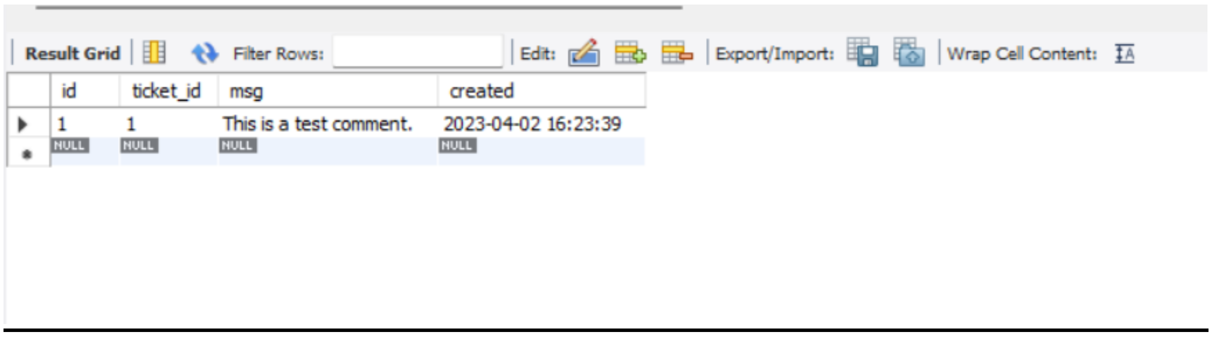
```
INSERT INTO `tickets` (`id`, `title`, `msg`, `email`, `created`, `status`) VALUES (1, 'Test Ticket', 'This is your first ticket.', 'support@codeshack.io', '2023-04-02 13:06:17', 'open');

CREATE TABLE IF NOT EXISTS `tickets_comments` (

        `id` int(11) NOT NULL AUTO_INCREMENT,

        `ticket_id` int(11) NOT NULL,

        `msg` text NOT NULL,

        `created` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,

        PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

INSERT INTO `tickets_comments` (`id`, `ticket_id`, `msg`, `created`) VALUES (1, 1, 'This is a test comment.', '2023-04-02 16:23:39');
```

OUTPUT OF TICKETS



id	title	msg	email	created	status
1	Test Ticket	This is your first ticket.	support@codeshack.io	2023-04-02 13:06:17	open
NULL	NULL	NULL	NULL	NULL	NULL

OUTPUT OF COMMENTS



id	ticket_id	msg	created
1	1	This is a test comment.	2023-04-02 16:23:39
NULL	NULL	NULL	NULL

Experiment-4

Preparing a greeting card and birthday card

Creating a database schema for a greeting card and birthday card preparation system involves organizing information about cards, customers, orders, and payments. Here's a basic example schema:

Cards Table: Stores information about the available cards.

```
CREATE TABLE Cards
(
   CardID INT PRIMARY KEY,
   Title VARCHAR(255),
   Description TEXT,
   Type VARCHAR(50) -- 'Greeting' or 'Birthday'
);
```

Customers Table: Stores information about customers who purchase cards.

```
CREATE TABLE Customers
(
   CustomerID INT PRIMARY KEY,
   Name VARCHAR(255),
   Email VARCHAR(255),
   Address VARCHAR(255)
);
```

Orders Table: Stores information about orders placed by customers.

```
CREATE TABLE Orders
(
   OrderID INT PRIMARY KEY,
   CustomerID INT,
   OrderDate DATE,
   TotalAmount DECIMAL(10, 2),
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

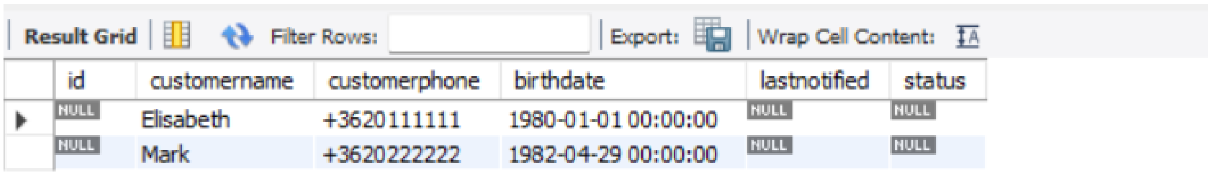
OrderItems Table: Stores information about individual card items within orders.

```
CREATE TABLE OrderItems
(
    OrderItemID INT PRIMARY KEY,
    OrderID INT,
    CardID INT,
    Quantity INT,
    Price DECIMAL(10, 2),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (CardID) REFERENCES Cards(CardID)
);
```

Payments Table: Stores information about payments made for orders.

```
CREATE TABLE Payments
(
    PaymentID INT PRIMARY KEY,
    OrderID INT,
    Amount DECIMAL(10, 2),
    PaymentDate DATE,
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)
);
```

OUTPUT



id	customername	customerphone	birthdate	lastnotified	status
NULL	Elisabeth	+3620111111	1980-01-01 00:00:00	NULL	NULL
NULL	Mark	+3620222222	1982-04-29 00:00:00	NULL	NULL

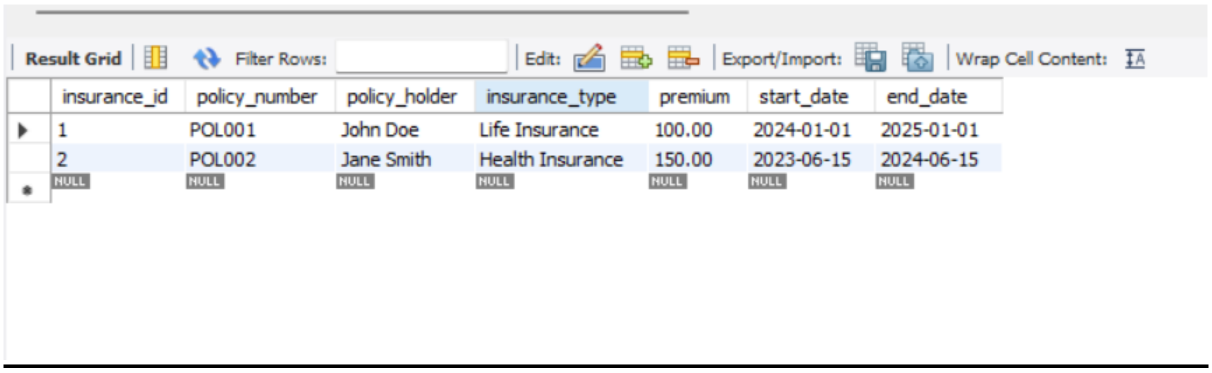
Experiment-05

Personal accounts - insurance,loans,mortgage payments dbms

Designing a database schema for managing personal accounts, including insurance, loans, and mortgage payments, involves organizing data related to accounts, transactions, payments, and possibly beneficiaries or policyholders. Here's a simplified example schema:

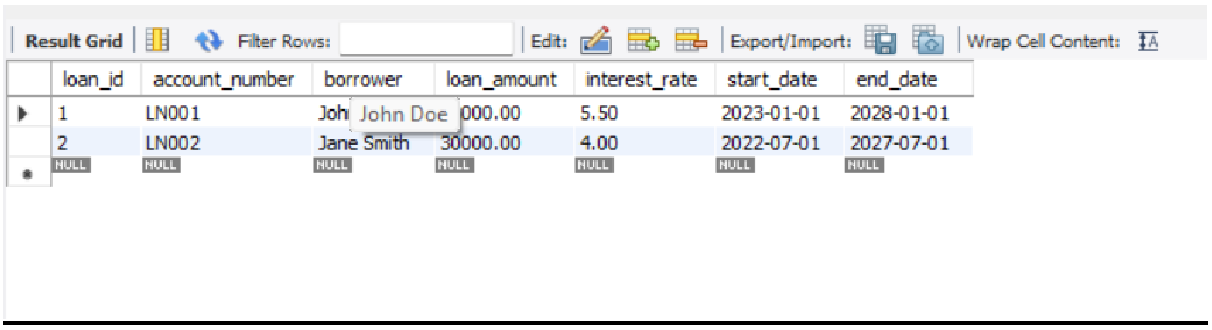
```
CREATE DATABASE IF NOT EXISTS personal_accounts;

USE personal_accounts;

CREATE TABLE insurance (
   insurance_id INT AUTO_INCREMENT PRIMARY KEY,
   policy_number VARCHAR(50) NOT NULL,
   policy_holder VARCHAR(100) NOT NULL,
   insurance_type VARCHAR(50) NOT NULL,
   premium DECIMAL(10, 2) NOT NULL,
   start_date DATE,
   end_date DATE,
   UNIQUE (policy_number)
);
CREATE TABLE loans (
   loan_id INT AUTO_INCREMENT PRIMARY KEY,
   account_number VARCHAR(50) NOT NULL,
   borrower VARCHAR(100) NOT NULL,
   loan_amount DECIMAL(12, 2) NOT NULL,
   interest_rate DECIMAL(5, 2) NOT NULL,
   start_date DATE,
   end_date DATE,
   UNIQUE (account_number)
);
CREATE TABLE mortgage_payments (
   payment_id INT AUTO_INCREMENT PRIMARY KEY,
   loan_id INT NOT NULL,
   payment_date DATE NOT NULL,
    amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (loan_id) REFERENCES loans(loan_id)
);
INSERT INTO insurance (policy_number, policy_holder, insurance_type, premium,
start_date, end_date)
VALUES
    ('POL001', 'John Doe', 'Life Insurance', 100.00, '2024-01-01', '2025-01-01'),
    ('POL002', 'Jane Smith', 'Health Insurance', 150.00, '2023-06-15', '2024-06-15');
INSERT INTO loans (account_number, borrower, loan_amount, interest_rate, start_date,
end_date)
VALUES
    ('LN001', 'John Doe', 50000.00, 5.5, '2023-01-01', '2028-01-01'),
    ('LN002', 'Jane Smith', 30000.00, 4.0, '2022-07-01', '2027-07-01');
INSERT INTO mortgage_payments (loan_id, payment_date, amount)
VALUES
    (1, '2024-07-01', 1000.00),
    (1, '2024-08-01', 1000.00),
    (2, '2024-07-15', 800.00);
SELECT * FROM insurance;
SELECT * FROM loans;
SELECT * FROM mortgage_payments;
```

<u>OUTPUT</u> SELECT * FROM insurance;



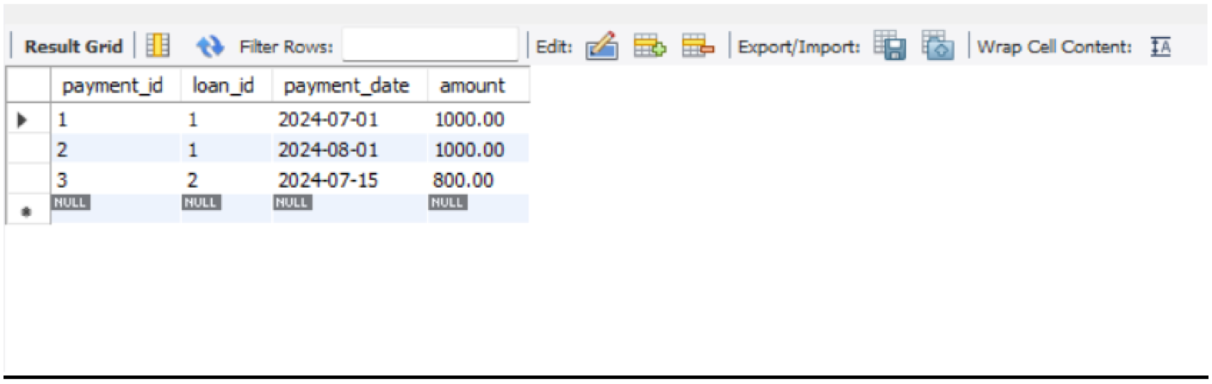
insurance_id	policy_number	policy_holder	insurance_type	premium	start_date	end_date
1	POL001	John Doe	Life Insurance	100.00	2024-01-01	2025-01-01
2	POL002	Jane Smith	Health Insurance	150.00	2023-06-15	2024-06-15
NULL	NULL	NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM loans;


Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

loan_id	account_number	borrower	loan_amount	interest_rate	start_date	end_date
1	LN001	John Doe	000.00	5.50	2023-01-01	2028-01-01
2	LN002	Jane Smith	30000.00	4.00	2022-07-01	2027-07-01
NULL	NULL	NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM mortgage_payments;


Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

payment_id	loan_id	payment_date	amount
1	1	2024-07-01	1000.00
2	1	2024-08-01	1000.00
3	2	2024-07-15	800.00
NULL	NULL	NULL	NULL

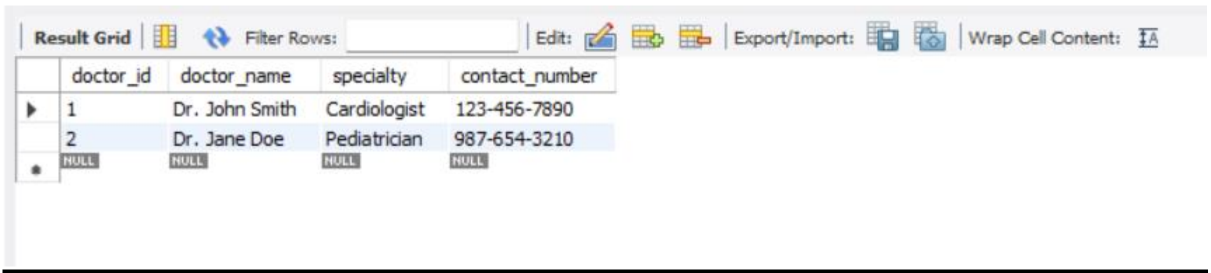
Experiment-06

Doctors diary and billing system

Designing a database schema for a doctor's diary and billing system involves organizing information about appointments, patients, medical records, invoices, and payments. Here's a basic example schema:

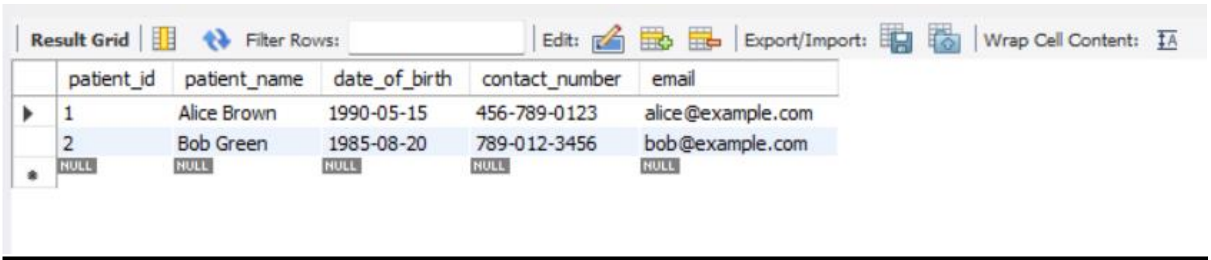
```
CREATE DATABASE IF NOT EXISTS doctors_diary_billing;

USE doctors_diary_billing;

CREATE TABLE doctors (

    doctor_id INT AUTO_INCREMENT PRIMARY KEY,

    doctor_name VARCHAR(100) NOT NULL,

    specialty VARCHAR(100),

    contact_number VARCHAR(20)

);

CREATE TABLE patients (

    patient_id INT AUTO_INCREMENT PRIMARY KEY,

    patient_name VARCHAR(100) NOT NULL,
    date_of_birth DATE,
    contact_number VARCHAR(20),
    email VARCHAR(100)
);
CREATE TABLE appointments (
    appointment_id INT AUTO_INCREMENT PRIMARY KEY,
    doctor_id INT NOT NULL,
    patient_id INT NOT NULL,
    appointment_date DATE NOT NULL,
    appointment_time TIME NOT NULL,
    status ENUM('Scheduled', 'Cancelled', 'Completed') DEFAULT 'Scheduled',
    FOREIGN KEY (doctor_id) REFERENCES doctors(doctor_id),
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id)
);
CREATE TABLE services (
    service_id INT AUTO_INCREMENT PRIMARY KEY,
    service_name VARCHAR(100) NOT NULL,
    service_fee DECIMAL(10, 2) NOT NULL
);
CREATE TABLE billing (
    billing_id INT AUTO_INCREMENT PRIMARY KEY,
    appointment_id INT NOT NULL,
    service_id INT NOT NULL,
    amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (appointment_id) REFERENCES appointments(appointment_id),
    FOREIGN KEY (service_id) REFERENCES services(service_id)
);
INSERT INTO doctors (doctor_name, specialty, contact_number)
VALUES
    ('Dr. John Smith', 'Cardiologist', '123-456-7890'),
    ('Dr. Jane Doe', 'Pediatrician', '987-654-3210');
INSERT INTO patients (patient_name, date_of_birth, contact_number, email)
VALUES
    ('Alice Brown', '1990-05-15', '456-789-0123', 'alice@example.com'),
    ('Bob Green', '1985-08-20', '789-012-3456', 'bob@example.com');
INSERT INTO appointments (doctor_id, patient_id, appointment_date, appointment_time)
VALUES
    (1, 1, '2024-07-16', '09:00:00'),
    (2, 2, '2024-07-17', '10:30:00');
INSERT INTO services (service_name, service_fee)
VALUES
    ('Consultation', 100.00),
    ('Check-up', 80.00);
INSERT INTO billing (appointment_id, service_id, amount)
VALUES
    (1, 1, 100.00),
    (2, 2, 80.00);
SELECT * FROM doctors;
SELECT * FROM patients;
SELECT * FROM appointments;
SELECT * FROM services;
SELECT b.billing_id, p.patient_name, d.doctor_name, s.service_name, s.service_fee,
b.amount
FROM billing b
JOIN appointments a ON b.appointment_id = a.appointment_id
JOIN patients p ON a.patient_id = p.patient_id
JOIN doctors d ON a.doctor_id = d.doctor_id
JOIN services s ON b.service_id = s.service_id;
```

<u>OUTPUT</u> SELECT * FROM doctors;



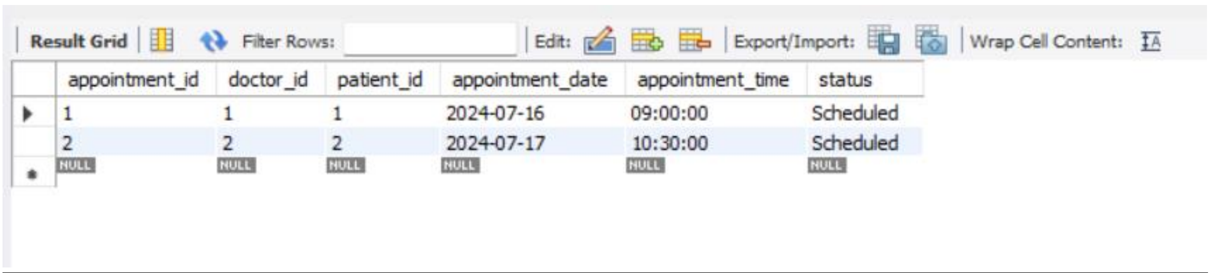
	doctor_id	doctor_name	specialty	contact_number
▶	1	Dr. John Smith	Cardiologist	123-456-7890
	2	Dr. Jane Doe	Pediatrician	987-654-3210
*	NULL	NULL	NULL	NULL

<u>OUTPUT</u> SELECT * FROM patients;



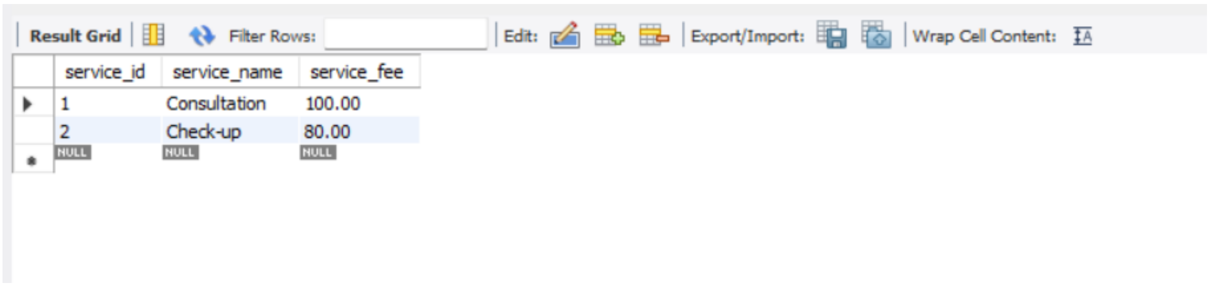
	patient_id	patient_name	date_of_birth	contact_number	email
▶	1	Alice Brown	1990-05-15	456-789-0123	alice@example.com
	2	Bob Green	1985-08-20	789-012-3456	bob@example.com
*	NULL	NULL	NULL	NULL	NULL

<u>OUTPUT</u> SELECT * FROM appointments;



	appointment_id	doctor_id	patient_id	appointment_date	appointment_time	status
▶	1	1	1	2024-07-16	09:00:00	Scheduled
	2	2	2	2024-07-17	10:30:00	Scheduled
*	NULL	NULL	NULL	NULL	NULL	NULL

<u>OUTPUT</u> SELECT * FROM services;



	service_id	service_name	service_fee
▶	1	Consultation	100.00
	2	Check-up	80.00
*	NULL	NULL	NULL

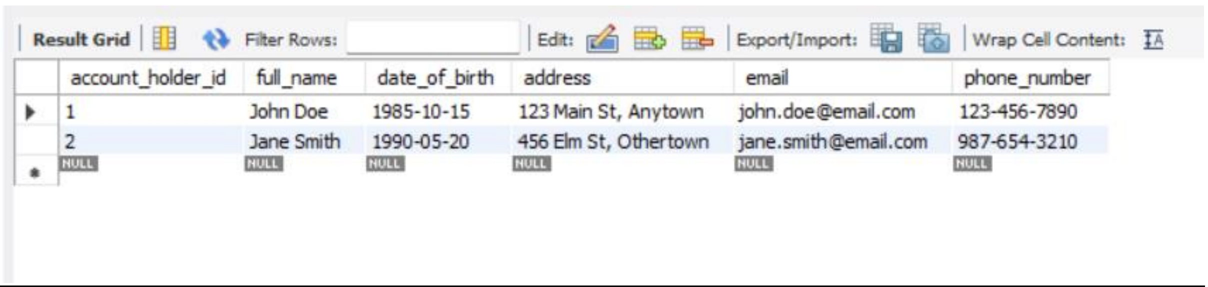
Experiment-07

Personal bank accounts

Designing a database schema for personal bank accounts involves organizing information about accounts, transactions, customers, and possibly other entities such as branches or employees. Here's a basic example schema:

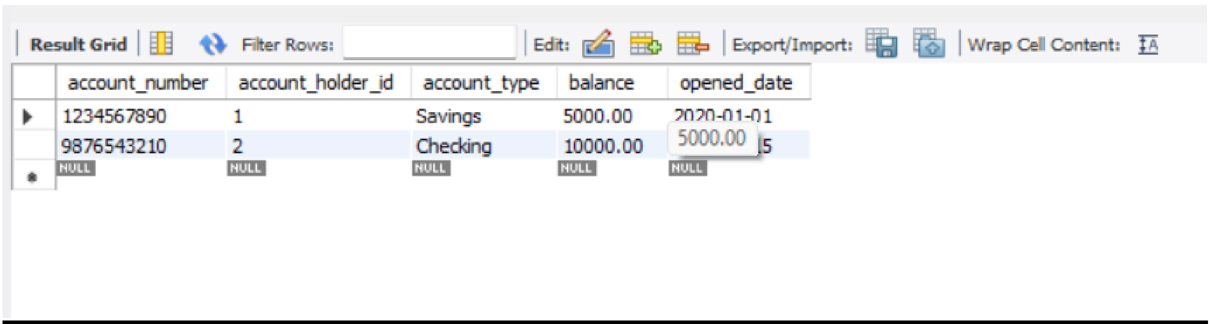
```
CREATE DATABASE IF NOT EXISTS personal_bank_accounts;
USE personal_bank_accounts;
CREATE TABLE account_holders (
    account_holder_id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    date_of_birth DATE,
    address VARCHAR(255),
    email VARCHAR(100),
    phone_number VARCHAR(20)
);
CREATE TABLE bank_accounts (
    account_number VARCHAR(50) PRIMARY KEY,
    account_holder_id INT NOT NULL,
    account_type VARCHAR(50) NOT NULL,
    balance DECIMAL(15, 2) NOT NULL,
    opened_date DATE,
    FOREIGN KEY (account_holder_id) REFERENCES account_holders(account_holder_id)
);
CREATE TABLE transactions (
    transaction_id INT AUTO_INCREMENT PRIMARY KEY,
    account_number VARCHAR(50) NOT NULL,
    transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    amount DECIMAL(10, 2) NOT NULL,
    transaction_type ENUM('Deposit', 'Withdrawal', 'Transfer') NOT NULL,
    FOREIGN KEY (account_number) REFERENCES bank_accounts(account_number)
);
INSERT INTO account_holders (full_name, date_of_birth, address, email, phone_number)
VALUES
    ('John Doe', '1985-10-15', '123 Main St, Anytown', 'john.doe@email.com', '123-456-7890'),
    ('Jane Smith', '1990-05-20', '456 Elm St, Othertown', 'jane.smith@email.com', '987-654-3210');
INSERT INTO bank_accounts (account_number, account_holder_id, account_type, balance, opened_date)
VALUES
    ('1234567890', 1, 'Savings', 5000.00, '2020-01-01'),
    ('9876543210', 2, 'Checking', 10000.00, '2018-05-15');
INSERT INTO transactions (account_number, amount, transaction_type)
VALUES
    ('1234567890', 1000.00, 'Deposit'),
    ('9876543210', 500.00, 'Withdrawal'),
    ('1234567890', 200.00, 'Transfer');
SELECT * FROM account_holders;
SELECT * FROM bank_accounts;
SELECT * FROM transactions;
```

OUTPUT SELECT * FROM account_holders;



account_holder_id	full_name	date_of_birth	address	email	phone_number
1	John Doe	1985-10-15	123 Main St, Anytown	john.doe@email.com	123-456-7890
2	Jane Smith	1990-05-20	456 Elm St, Othertown	jane.smith@email.com	987-654-3210
NULL	NULL	NULL	NULL	NULL	NULL

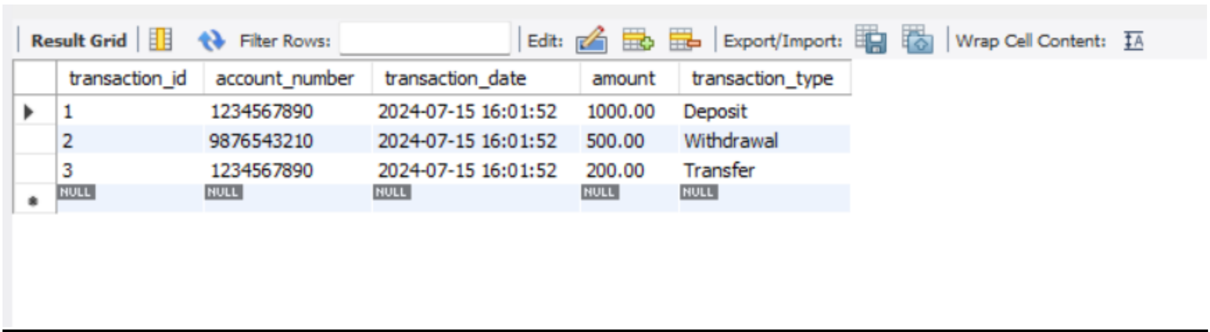
OUTPUT SELECT * FROM bank_accounts;



account_number	account_holder_id	account_type	balance	opened_date
1234567890	1	Savings	5000.00	2020-01-01
9876543210	2	Checking	10000.00	[illegible]5
NULL	NULL	NULL	NULL	NULL

5000.00

OUTPUT SELECT * FROM transactions;



transaction_id	account_number	transaction_date	amount	transaction_type
1	1234567890	2024-07-15 16:01:52	1000.00	Deposit
2	9876543210	2024-07-15 16:01:52	500.00	Withdrawal
3	1234567890	2024-07-15 16:01:52	200.00	Transfer
NULL	NULL	NULL	NULL	NULL

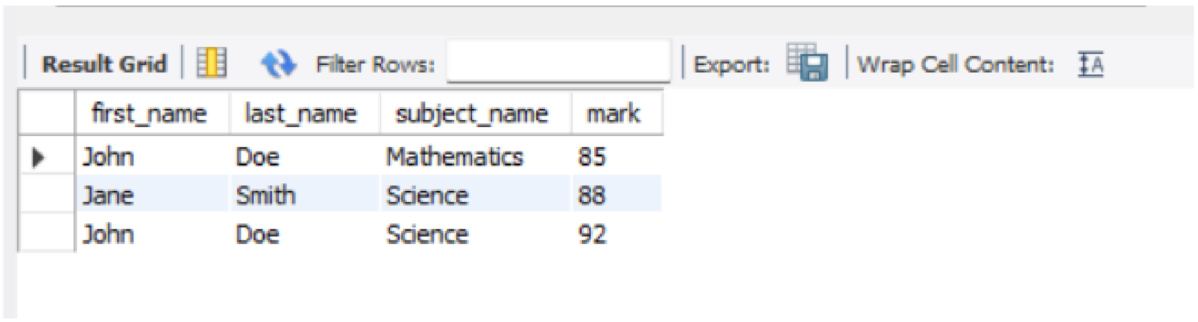
Experiment-08

Class marks management's.

To manage class marks in a SQL database, you'll typically need at least two tables: one for storing information about students and another for storing their marks.

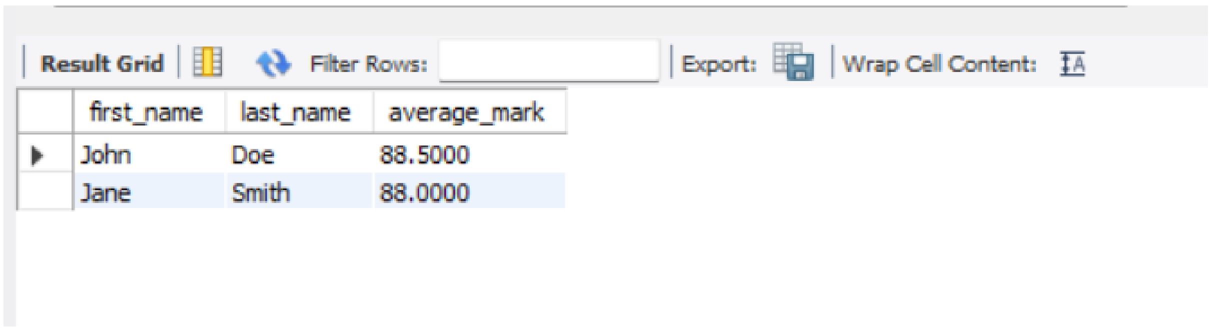
```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE
);
CREATE TABLE subjects (
    subject_id INT AUTO_INCREMENT PRIMARY KEY,
    subject_name VARCHAR(100)
);
CREATE TABLE marks (
    mark_id INT AUTO_INCREMENT PRIMARY KEY,
    student_id INT,
    subject_id INT,
    mark INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (subject_id) REFERENCES subjects(subject_id)
);
INSERT INTO students (first_name, last_name, date_of_birth) VALUES
('John', 'Doe', '2000-01-01'),
('Jane', 'Smith', '2001-02-02');
INSERT INTO subjects (subject_name) VALUES
('Mathematics'),
('Science');
INSERT INTO marks (student_id, subject_id, mark) VALUES
(1, 1, 85),
(1, 2, 90),
(2, 1, 78),
(2, 2, 88);
SELECT
    students.first_name,
    students.last_name,
    subjects.subject_name,
    marks.mark
FROM
    marks
JOIN
    students ON marks.student_id = students.student_id
JOIN
    subjects ON marks.subject_id = subjects.subject_id;
SELECT
    students.first_name,
    students.last_name,
    AVG(marks.mark) AS average_mark
FROM
    marks
JOIN
    students ON marks.student_id = students.student_id
GROUP BY
    students.student_id;
UPDATE marks
SET mark = 92
WHERE student_id = 1 AND subject_id = 2;
DELETE FROM marks
WHERE student_id = 2 AND subject_id = 1;
```

OUTPUT



first_name	last_name	subject_name	mark
John	Doe	Mathematics	85
Jane	Smith	Science	88
John	Doe	Science	92

OUTPUT



first_name	last_name	average_mark
John	Doe	88.5000
Jane	Smith	88.0000

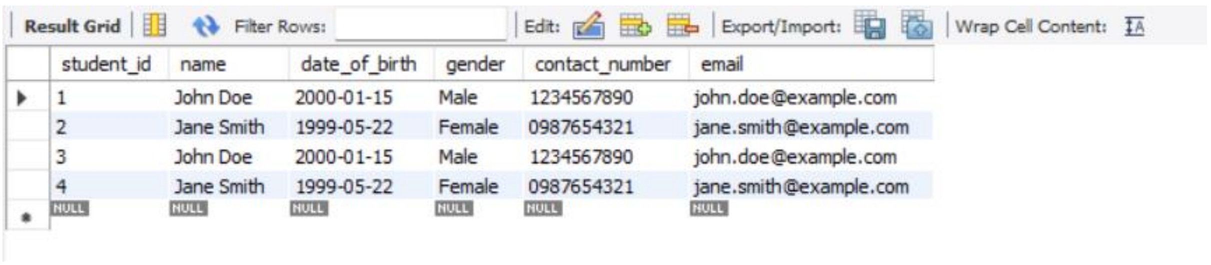
Experiment-9

Hostel accounting

Managing hostel accounting in a SQL database involves tracking various aspects such as room allocation, expenses, payments, and resident details. Here's a basic example schema:

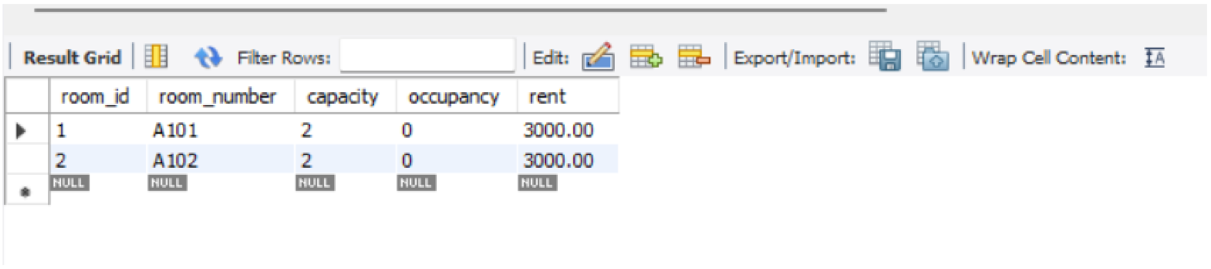
```
CREATE TABLE Students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    NAMES VARCHAR(100),
    date_of_birth DATE,
    gender ENUM('Male', 'Female', 'Other'),
    contact_number VARCHAR(15),
    email VARCHAR(100)
);
CREATE TABLE Rooms (
    room_id INT AUTO_INCREMENT PRIMARY KEY,
    room_number VARCHAR(10) UNIQUE,
    capacity INT,
    occupancy INT DEFAULT 0,
    rent DECIMAL(10, 2)
);
CREATE TABLE Transactions (
    transaction_id INT AUTO_INCREMENT PRIMARY KEY,
    student_id INT,
    amount DECIMAL(10, 2),
    transaction_date DATE,
    description VARCHAR(255),
    FOREIGN KEY (student_id) REFERENCES Students(student_id)
);
CREATE TABLE Expenses (
    expense_id INT AUTO_INCREMENT PRIMARY KEY,
    amount DECIMAL(10, 2),
    expense_date DATE,
    description VARCHAR(255)
);
INSERT INTO Students (name, date_of_birth, gender, contact_number, email) VALUES
('John Doe', '2000-01-15', 'Male', '1234567890', 'john.doe@example.com'),
('Jane Smith', '1999-05-22', 'Female', '0987654321', 'jane.smith@example.com');
INSERT INTO Rooms (room_number, capacity, rent) VALUES
('A101', 2, 3000.00),
('A102', 2, 3000.00);
INSERT INTO Transactions (student_id, amount, transaction_date, description) VALUES
(1, 3000.00, '2024-07-01', 'Monthly Rent'),
(2, 3000.00, '2024-07-01', 'Monthly Rent');
INSERT INTO Expenses (amount, expense_date, description) VALUES
(1500.00, '2024-07-05', 'Electricity Bill'),
(500.00, '2024-07-10', 'Water Bill');
SELECT * FROM Students;
SELECT * FROM Rooms;
SELECT * FROM Transactions;
SELECT * FROM Expenses;
SELECT SUM(amount) AS TotalIncome FROM Transactions;
SELECT SUM(amount) AS TotalExpenses FROM Expenses;
SELECT
    (SELECT SUM(amount) FROM Transactions) -
    (SELECT SUM(amount) FROM Expenses) AS NetIncome;
```

OUTPUT SELECT * FROM Students;



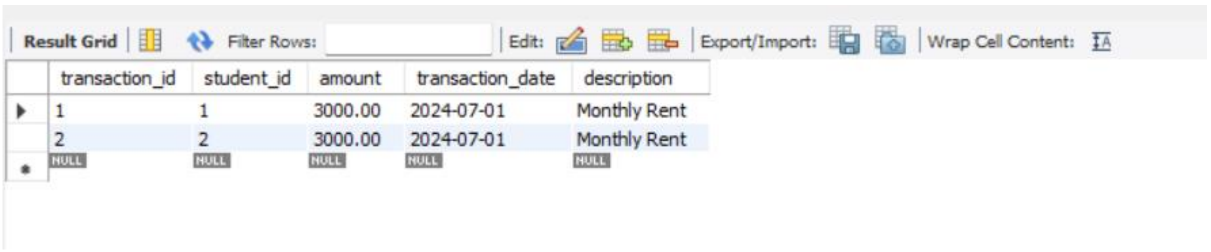
student_id	name	date_of_birth	gender	contact_number	email
1	John Doe	2000-01-15	Male	1234567890	john.doe@example.com
2	Jane Smith	1999-05-22	Female	0987654321	jane.smith@example.com
3	John Doe	2000-01-15	Male	1234567890	john.doe@example.com
4	Jane Smith	1999-05-22	Female	0987654321	jane.smith@example.com
NULL	NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM Rooms;



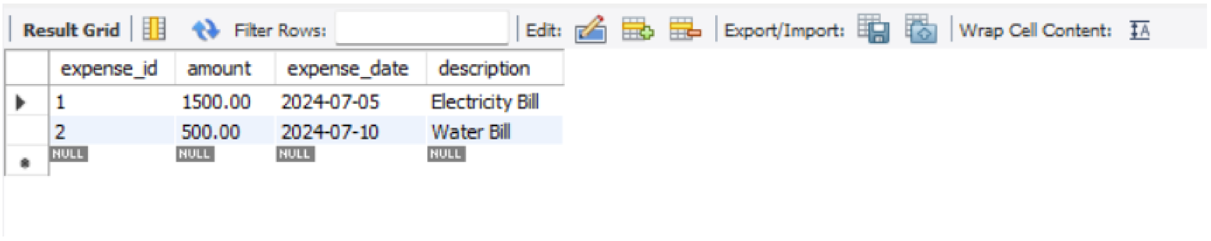
room_id	room_number	capacity	occupancy	rent
1	A101	2	0	3000.00
2	A102	2	0	3000.00
NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM Transactions;



transaction_id	student_id	amount	transaction_date	description
1	1	3000.00	2024-07-01	Monthly Rent
2	2	3000.00	2024-07-01	Monthly Rent
NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM Expenses;



expense_id	amount	expense_date	description
1	1500.00	2024-07-05	Electricity Bill
2	500.00	2024-07-10	Water Bill
NULL	NULL	NULL	NULL

Experiment-10

Video tape library

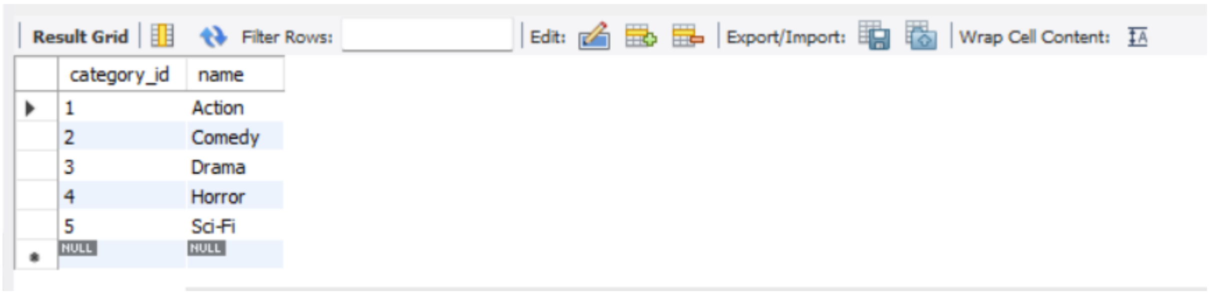
Managing a video tape library in a SQL database involves keeping track of various details about the tapes, such as their titles, genres, availability, and borrower information.

```
CREATE TABLE Categories (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) UNIQUE
);
CREATE TABLE Tapes (
    tape_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    category_id INT,
    release_year YEAR,
    stock INT,
    FOREIGN KEY (category_id) REFERENCES Categories(category_id)
);
CREATE TABLE Members (
    member_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    date_of_birth DATE,
    contact_number VARCHAR(15),
    email VARCHAR(100)
);
CREATE TABLE Rentals (
    rental_id INT AUTO_INCREMENT PRIMARY KEY,
    tape_id INT,
    member_id INT,
    rental_date DATE,
    return_date DATE,
    FOREIGN KEY (tape_id) REFERENCES Tapes(tape_id),
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
INSERT INTO Categories (name) VALUES
('Action'),
('Comedy'),
('Drama'),
('Horror'),
('Sci-Fi');
INSERT INTO Tapes (title, category_id, release_year, stock) VALUES
('Die Hard', 1, 1988, 5),
('The Mask', 2, 1994, 3),
('The Godfather', 3, 1972, 2),
('The Exorcist', 4, 1973, 4),
('Star Wars', 5, 1977, 6);
INSERT INTO Members (name, date_of_birth, contact_number, email) VALUES
('John Doe', '1980-05-15', '1234567890', 'john.doe@example.com'),
('Jane Smith', '1992-11-22', '0987654321', 'jane.smith@example.com');
INSERT INTO Rentals (tape_id, member_id, rental_date, return_date) VALUES
(1, 1, '2024-07-01', '2024-07-07'),
(2, 2, '2024-07-03', '2024-07-10');
SELECT * FROM Categories;
SELECT * FROM Tapes;
SELECT * FROM Members;
SELECT * FROM Rentals;
SELECT
    c.name AS Category,
    COUNT(r.rental_id) AS TotalRentals
FROM
    Rentals r
JOIN
```

```
    Tapes t ON r.tape_id = t.tape_id
JOIN
    Categories c ON t.category_id = c.category_id
GROUP BY
    c.name;
SELECT
    t.title,
    COUNT(r.rental_id) AS RentalCount
FROM
    Rentals r
JOIN
    Tapes t ON r.tape_id = t.tape_id
GROUP BY
    t.title
ORDER BY
    RentalCount DESC
LIMIT 5;
SELECT
    m.name,
    m.contact_number,
    m.email,
    t.title,
    r.rental_date,
    r.return_date
FROM
    Rentals r
JOIN
    Members m ON r.member_id = m.member_id
JOIN
    Tapes t ON r.tape_id = t.tape_id
```

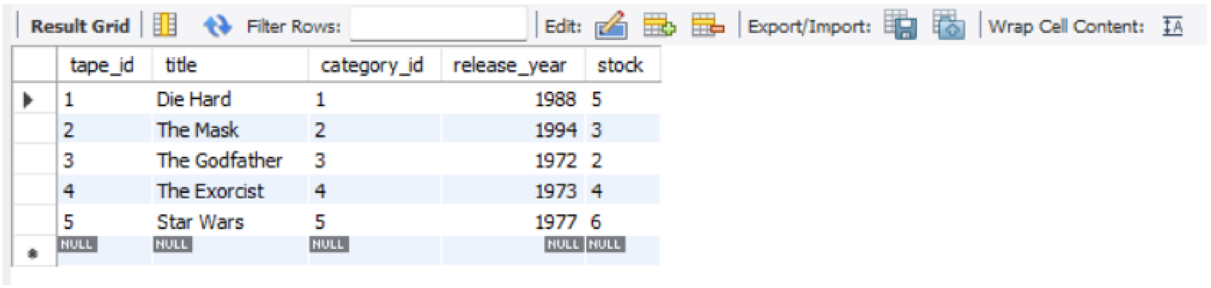
```
WHERE
    r.return_date < CURDATE() AND r.return_date IS NOT NULL;
```

OUTPUT SELECT * FROM Categories;



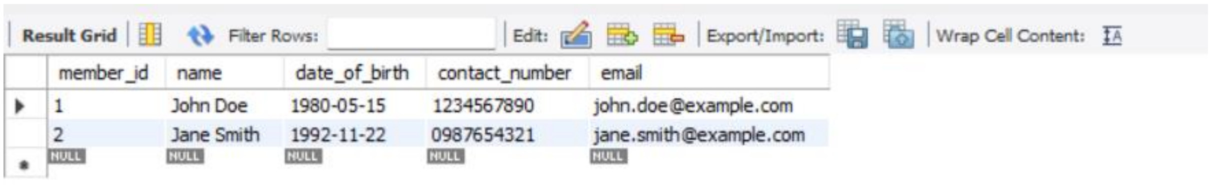
category_id	name
1	Action
2	Comedy
3	Drama
4	Horror
5	Sci-Fi
NULL	NULL

OUTPUT SELECT * FROM Tapes;



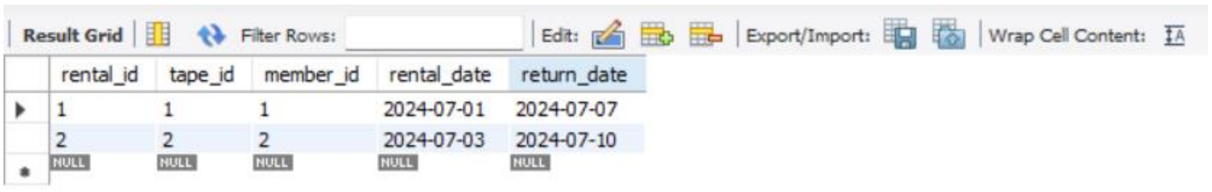
tape_id	title	category_id	release_year	stock
1	Die Hard	1	1988	5
2	The Mask	2	1994	3
3	The Godfather	3	1972	2
4	The Exorcist	4	1973	4
5	Star Wars	5	1977	6
NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM Members;



member_id	name	date_of_birth	contact_number	email
1	John Doe	1980-05-15	1234567890	john.doe@example.com
2	Jane Smith	1992-11-22	0987654321	jane.smith@example.com
NULL	NULL	NULL	NULL	NULL

OUTPUT SELECT * FROM Rentals;



rental_id	tape_id	member_id	rental_date	return_date
1	1	1	2024-07-01	2024-07-07
2	2	2	2024-07-03	2024-07-10
NULL	NULL	NULL	NULL	NULL

Experiment-11

History of Cricket Scores

Early Cricket Scoring (16th to 18th Century)

1. First Scoring Methods:
   - Cricket originated in England in the 16th century.
   - In the early days, scoring was primitive. Players or umpires used notches on sticks or tally sticks to record runs.
   - Each run was represented by a single notch.
2. Introduction of Written Scores:
   - In the late 18th century, scorecards were first introduced, allowing scores to be recorded on paper.
   - The first recorded cricket match scorecard is believed to date to 1776, for a match at the Vine Cricket Ground in Kent.

19th Century Developments

1. Scorecards Standardized:
   - By the mid-19th century, printed scorecards became standard at matches.
   - Lord's Cricket Ground began distributing pre-printed scorecards in 1846, which marked a significant step in formalizing scorekeeping.
2. Scorers and Symbols:
   - Dedicated scorers started recording match events.
   - Symbols were introduced to represent different events in the game:
     - A dot (•) for a ball with no run (origin of the term "dot ball").
     - Numbers to indicate runs scored per ball.
     - Letters like "W" for wickets and "b" for byes.
3. Telegraph Boards:
   - Large telegraph boards were used to display scores for spectators at the ground.

20th Century Innovations

1. Manual Scoreboards:
   - Large manual scoreboards became popular, especially in Test cricket venues.

    - These boards displayed key details such as total runs, wickets, and overs bowled.
2. Advent of Broadcasting:
    - With radio and later television, scorers had to provide accurate and real-time updates for commentary.
    - This made scoring a specialized and professional role.
3. Limited-Overs Cricket:
    - The introduction of One-Day Internationals (ODIs) in the 1970s brought new scoring challenges, including recording strike rates and economy rates.
4. Electronic Scoreboards:
    - In the 1980s, electronic scoreboards started replacing manual ones, offering more detailed real-time information.

Key Milestones in Cricket Scoring History

- **1776**: First known scorecard used in England.
- **1846**: Standardized printed scorecards introduced at Lord's.
- **1899**: First use of telegraph boards at cricket grounds.
- **1980s**: Electronic scoreboards introduced.
- **1990s**: Computerized scoring systems implemented.
- **2000s**: Real-time online scoring and data analytics became common.

Experiment-12

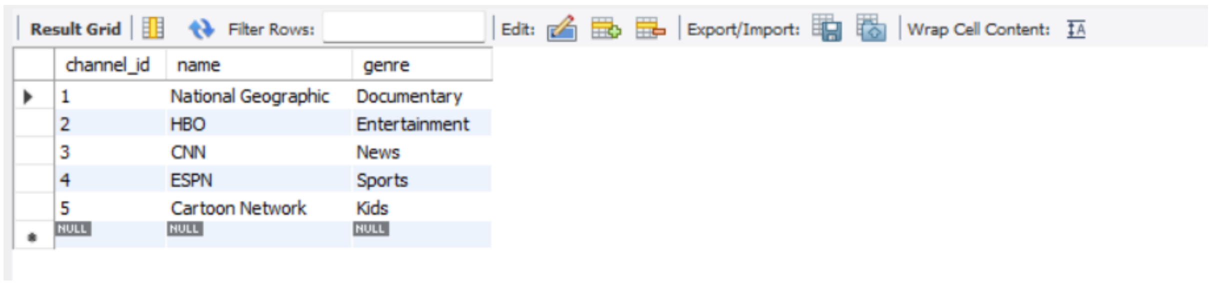
Cable TV transmission program manager

Managing a cable TV transmission program involves scheduling programs, managing channels, and keeping track of program details. Below is a basic schema for such a system:

```
CREATE TABLE Channels (
    channel_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) UNIQUE,
    genre VARCHAR(50)
);
CREATE TABLE Programs (
    program_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    description TEXT,
    duration INT,  -- duration in minutes
    channel_id INT,
    FOREIGN KEY (channel_id) REFERENCES Channels(channel_id)
);
CREATE TABLE Schedules (
    schedule_id INT AUTO_INCREMENT PRIMARY KEY,
    program_id INT,
    start_time DATETIME,
    end_time DATETIME,
    FOREIGN KEY (program_id) REFERENCES Programs(program_id)
);
CREATE TABLE Viewers (
    viewer_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    date_of_birth DATE,
    contact_number VARCHAR(15),
    email VARCHAR(100)
);
CREATE TABLE Subscriptions (
    subscription_id INT AUTO_INCREMENT PRIMARY KEY,
    viewer_id INT,
    channel_id INT,
    subscription_date DATE,
    FOREIGN KEY (viewer_id) REFERENCES Viewers(viewer_id),
    FOREIGN KEY (channel_id) REFERENCES Channels(channel_id)
);
INSERT INTO Channels (name, genre) VALUES
('National Geographic', 'Documentary'),
('HBO', 'Entertainment'),
('CNN', 'News'),
('ESPN', 'Sports'),
('Cartoon Network', 'Kids');
INSERT INTO Programs (title, description, duration, channel_id) VALUES
('Wildlife Documentary', 'A documentary about wildlife.', 60, 1),
('Game of Thrones', 'Fantasy drama series.', 60, 2),
('Daily News', 'Latest news and updates.', 30, 3),
('Football Live', 'Live football match.', 120, 4),
('Cartoon Show', 'Popular cartoon series.', 30, 5);
INSERT INTO Schedules (program_id, start_time, end_time) VALUES
(1, '2024-07-15 08:00:00', '2024-07-15 09:00:00'),
(2, '2024-07-15 21:00:00', '2024-07-15 22:00:00'),
(3, '2024-07-15 18:00:00', '2024-07-15 18:30:00'),
(4, '2024-07-15 15:00:00', '2024-07-15 17:00:00'),
(5, '2024-07-15 10:00:00', '2024-07-15 10:30:00');
INSERT INTO Viewers (name, date_of_birth, contact_number, email) VALUES
('Alice Johnson', '1985-04-15', '1234567890', 'alice.johnson@example.com'),
('Bob Smith', '1990-07-22', '0987654321', 'bob.smith@example.com');
INSERT INTO Subscriptions (viewer_id, channel_id, subscription_date) VALUES
(1, 1, '2024-07-01'),
(1, 2, '2024-07-01'),
(2, 3, '2024-07-01'),
(2, 4, '2024-07-01');
SELECT * FROM Channels;
SELECT * FROM Programs;
SELECT * FROM Schedules;
SELECT * FROM Viewers;
SELECT * FROM Subscriptions;
SELECT
    s.schedule_id,
    p.title,
    c.name AS channel_name,
    s.start_time,
    s.end_time
FROM
    Schedules s
JOIN
    Programs p ON s.program_id = p.program_id
JOIN
    Channels c ON p.channel_id = c.channel_id
WHERE
    DATE(s.start_time) = '2024-07-15';
SELECT
    c.name AS channel_name,
    p.title,
    p.description,
    p.duration
FROM
    Programs p
JOIN
    Channels c ON p.channel_id = c.channel_id;
SELECT
    v.name AS viewer_name,
    c.name AS channel_name,
    s.subscription_date
FROM
    Subscriptions s
JOIN
    Viewers v ON s.viewer_id = v.viewer_id
JOIN
```

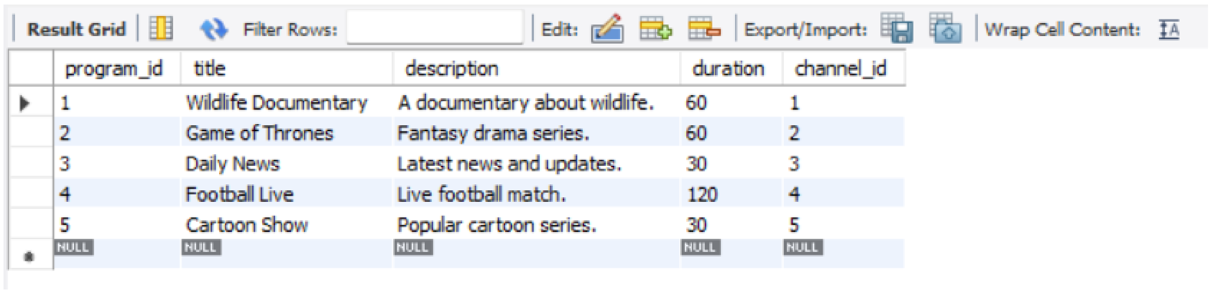
Channels c ON s.channel_id = c.channel_id;

Output SELECT * FROM Channels;



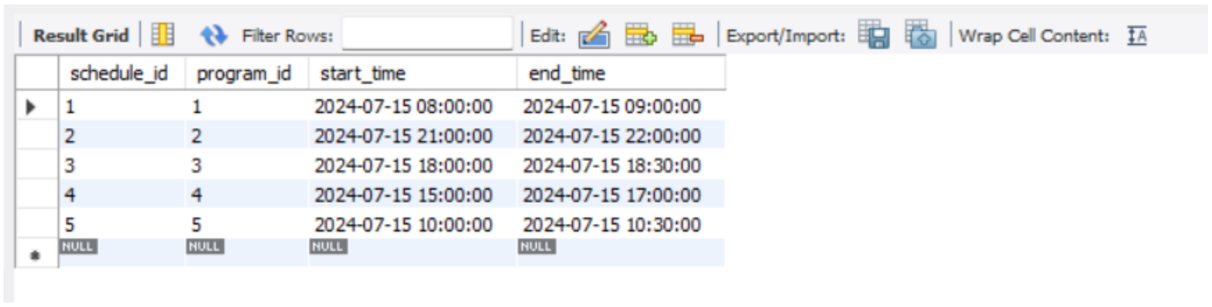
channel_id	name	genre
1	National Geographic	Documentary
2	HBO	Entertainment
3	CNN	News
4	ESPN	Sports
5	Cartoon Network	Kids
NULL	NULL	NULL

Output SELECT * FROM Programs;



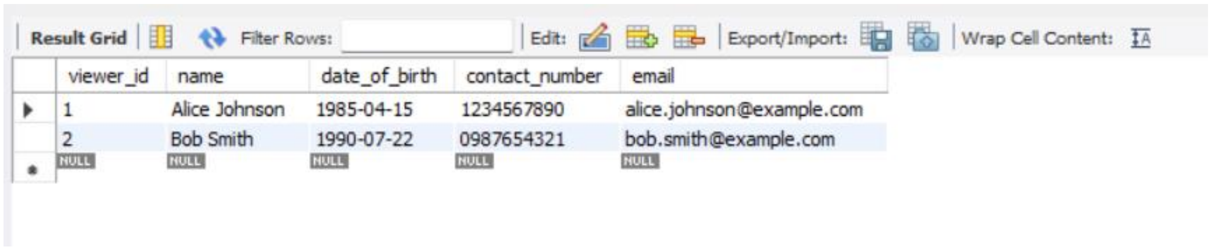
program_id	title	description	duration	channel_id
1	Wildlife Documentary	A documentary about wildlife.	60	1
2	Game of Thrones	Fantasy drama series.	60	2
3	Daily News	Latest news and updates.	30	3
4	Football Live	Live football match.	120	4
5	Cartoon Show	Popular cartoon series.	30	5
NULL	NULL	NULL	NULL	NULL

Output SELECT * FROM Schedules;



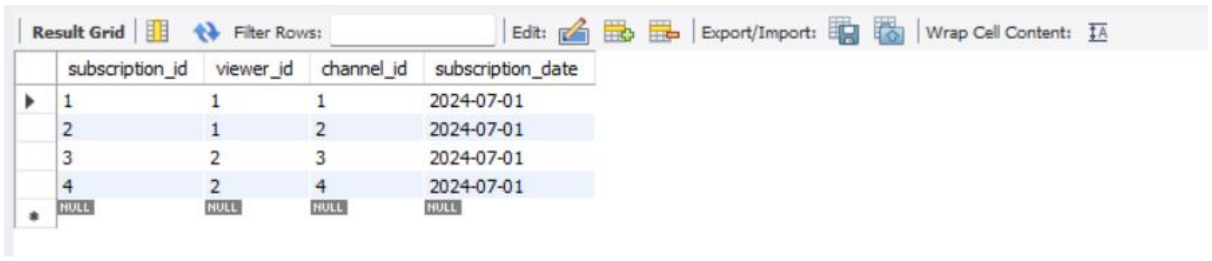
schedule_id	program_id	start_time	end_time
1	1	2024-07-15 08:00:00	2024-07-15 09:00:00
2	2	2024-07-15 21:00:00	2024-07-15 22:00:00
3	3	2024-07-15 18:00:00	2024-07-15 18:30:00
4	4	2024-07-15 15:00:00	2024-07-15 17:00:00
5	5	2024-07-15 10:00:00	2024-07-15 10:30:00
NULL	NULL	NULL	NULL

Output SELECT * FROM Viewers;



viewer_id	name	date_of_birth	contact_number	email
1	Alice Johnson	1985-04-15	1234567890	alice.johnson@example.com
2	Bob Smith	1990-07-22	0987654321	bob.smith@example.com
NULL	NULL	NULL	NULL	NULL

SELECT * FROM Subscriptions;



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

	subscription_id	viewer_id	channel_id	subscription_date
▶	1	1	1	2024-07-01
	2	1	2	2024-07-01
	3	2	3	2024-07-01
	4	2	4	2024-07-01
*	NULL	NULL	NULL	NULL

Experiment-13

Personal library

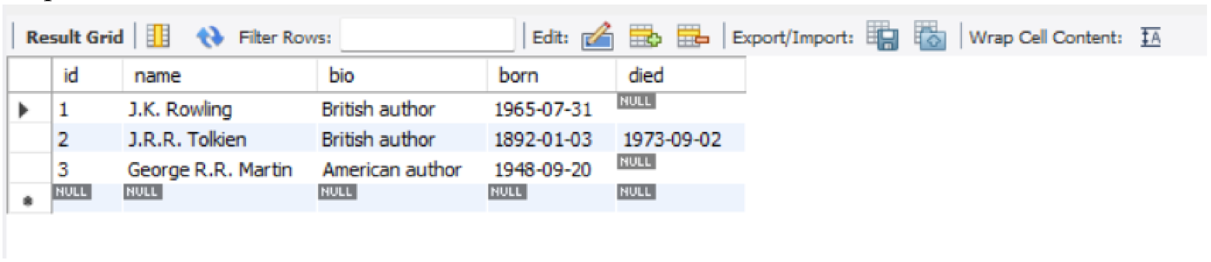
Managing a personal library in a SQL database involves keeping track of books, authors, genres, and borrowing history. Here's a basic schema to get you started:

```
CREATE TABLE books (
  `id` INT AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  `author` VARCHAR(100) NOT NULL,
  `publisher` VARCHAR(100) NOT NULL,
  `published_date` DATE NOT NULL,
  `genre` VARCHAR(50) NOT NULL,
  `summary` TEXT,
  `isbn` VARCHAR(20) NOT NULL,
  `edition` VARCHAR(20) NOT NULL,
  `pages` INT NOT NULL,
  `language` VARCHAR(20) NOT NULL,
  `format` VARCHAR(20) NOT NULL,
  `rating` DECIMAL(3,2) NOT NULL DEFAULT 0.00,
  `review` TEXT,
  PRIMARY KEY (`id`)
);
CREATE TABLE authors (
  `id` INT AUTO_INCREMENT,
  `name` VARCHAR(100) NOT NULL,
  `bio` TEXT,
  `born` DATE,
  `died` DATE,
  PRIMARY KEY (`id`)
);
CREATE TABLE book_categories (
  `id` INT AUTO_INCREMENT,
  `name` VARCHAR(50) NOT NULL,
  PRIMARY KEY (`id`)
);
CREATE TABLE book_category_mappings (
  `id` INT AUTO_INCREMENT,
  `book_id` INT NOT NULL,
  `category_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  FOREIGN KEY (`book_id`) REFERENCES books(`id`),
  FOREIGN KEY (`category_id`) REFERENCES book_categories(`id`)
);
CREATE TABLE book_shelves (
  `id` INT AUTO_INCREMENT,
  `name` VARCHAR(50) NOT NULL,
  PRIMARY KEY (`id`)
);
CREATE TABLE book_shelf_mappings (
  `id` INT AUTO_INCREMENT,
  `book_id` INT NOT NULL,
  `shelf_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  FOREIGN KEY (`book_id`) REFERENCES books(`id`),
  FOREIGN KEY (`shelf_id`) REFERENCES book_shelves(`id`)
);
```

```
CREATE TABLE borrowing_history (
  `id` INT AUTO_INCREMENT,
  `book_id` INT NOT NULL,
  `borrower_name` VARCHAR(100) NOT NULL,
  `borrow_date` DATE NOT NULL,
  `return_date` DATE,
  `is_returned` BOOLEAN NOT NULL DEFAULT FALSE,
  PRIMARY KEY (`id`),
  FOREIGN KEY (`book_id`) REFERENCES books(`id`)
);
INSERT INTO authors (name, bio, born, died) VALUES
  ('J.K. Rowling', 'British author', '1965-07-31', NULL),
  ('J.R.R. Tolkien', 'British author', '1892-01-03', '1973-09-02'),
  ('George R.R. Martin', 'American author', '1948-09-20', NULL);
INSERT INTO books (title, author, publisher, published_date, genre, summary, isbn, edition,
pages, language, format, rating, review) VALUES
  ('Harry Potter and the Philosopher\'s Stone', 'J.K. Rowling', 'Bloomsbury', '1997-06-26',
'Fantasy', 'The first book in the Harry Potter series', '9780747532743', 'First Edition', 320,
'English', 'Hardcover', 4.50, 'A great start to the series'),
  ('The Lord of the Rings', 'J.R.R. Tolkien', 'Allen & Unwin', '1954-07-29', 'Fantasy', 'The
classic high fantasy novel', '9780261103573', 'First Edition', 1216, 'English', 'Hardcover',
4.80, 'A classic'),
  ('A Game of Thrones', 'George R.R. Martin', 'Bantam Books', '1996-08-01', 'Fantasy', 'The
first book in the A Song of Ice and Fire series', '9780553103547', 'First Edition', 835,
'English', 'Hardcover', 4.60, 'A great start to the series');
INSERT INTO book_categories (name) VALUES
  ('Fantasy'),
  ('Science Fiction'),
  ('Romance');
INSERT INTO book_category_mappings (book_id, category_id) VALUES
  (1, 1),
  (2, 1),
  (3, 1);
```

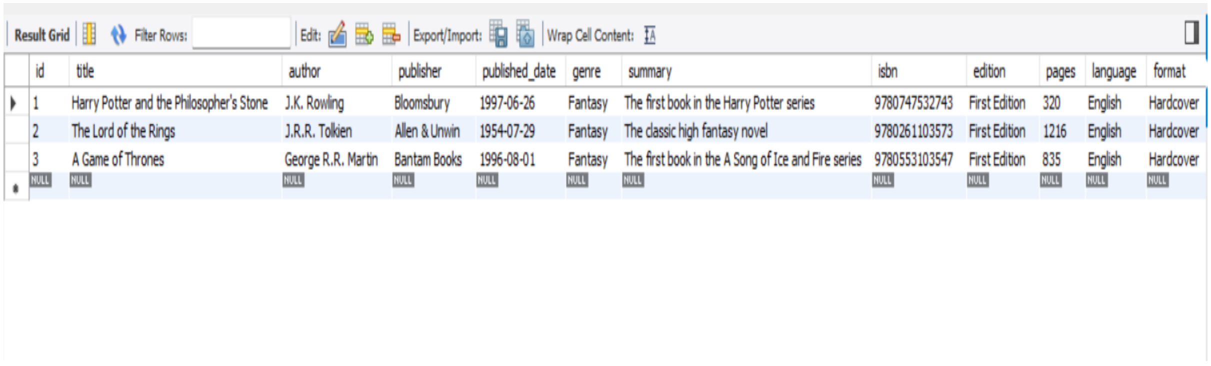
```
INSERT INTO book_shelves (name) VALUES
  ('Fiction'),
  ('Non-Fiction'),
  ('Biography');
INSERT INTO book_shelf_mappings (book_id, shelf_id) VALUES
  (1, 1),
  (2, 1),
  (3, 1);
INSERT INTO borrowing_history (book_id, borrower_name, borrow_date, return_date,
is_returned) VALUES
  (1, 'John Doe', '2022-01-01', '2022-01-15', TRUE),
  (2, 'Jane Doe', '2022-02-01', '2022-02-15', TRUE),
  (3, 'Bob Smith', '2022-03-01', '2022-03-15', TRUE);


  select * from authors;
    select * from books;
      select * from  book_categories;
        select * from book_category_mappings;
          select * from borrowing_history;
```

output select * from authors;



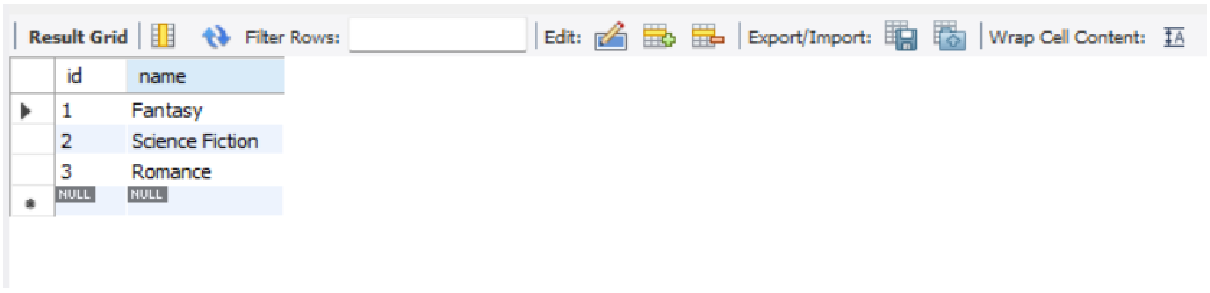
id	name	bio	born	died
1	J.K. Rowling	British author	1965-07-31	NULL
2	J.R.R. Tolkien	British author	1892-01-03	1973-09-02
3	George R.R. Martin	American author	1948-09-20	NULL
NULL	NULL	NULL	NULL	NULL

output select * from books;



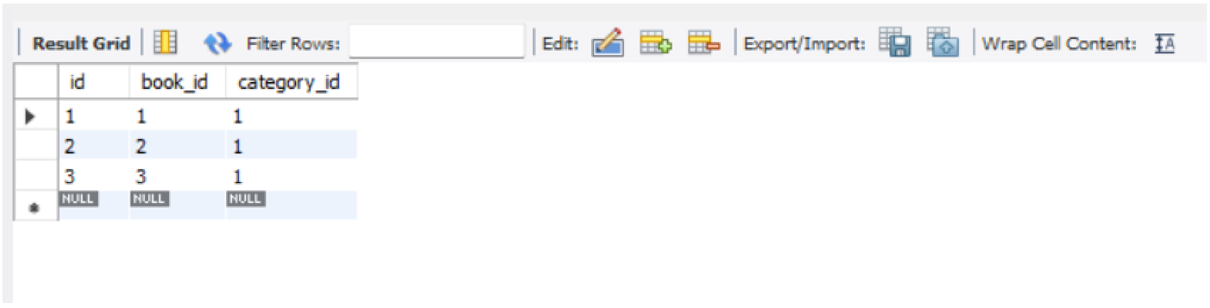
id	title	author	publisher	published_date	genre	summary	isbn	edition	pages	language	format
1	Harry Potter and the Philosopher's Stone	J.K. Rowling	Bloomsbury	1997-06-26	Fantasy	The first book in the Harry Potter series	9780747532743	First Edition	320	English	Hardcover
2	The Lord of the Rings	J.R.R. Tolkien	Allen & Unwin	1954-07-29	Fantasy	The classic high fantasy novel	9780261103573	First Edition	1216	English	Hardcover
3	A Game of Thrones	George R.R. Martin	Bantam Books	1996-08-01	Fantasy	The first book in the A Song of Ice and Fire series	9780553103547	First Edition	835	English	Hardcover
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

output select * from book_categories;



id	name
1	Fantasy
2	Science Fiction
3	Romance
NULL	NULL

output select * from book_category_mappings;



id	book_id	category_id
1	1	1
2	2	1
3	3	1
NULL	NULL	NULL

Experiment -14

Sailors Database Schema

1. Sailors Table

Details about sailors, including their unique ID, name, rating, and age.

Column	Data Type	Constraints
SailorID	INT	Primary Key
Name	VARCHAR(100)	Not Null
Rating	INT	Not Null
Age	DECIMAL(4,1)	Not Null

2. Boats Table

Details about boats, including their unique ID, name, color, and capacity.

Column	Data Type	Constraints
BoatID	INT	Primary Key
Name	VARCHAR(100)	Not Null
Color	VARCHAR(50)	Not Null
Capacity	INT	Not Null

3. Reservations Table

Tracks which sailor has reserved which boat on a specific date.

Column	Data Type	Constraints
ReservationID	INT	Primary Key
SailorID	INT	Foreign Key → Sailors
BoatID	INT	Foreign Key → Boats
ReservationDate	DATE	Not Null

SQL Implementation

```
-- Sailors Table
CREATE TABLE Sailors (
    SailorID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Rating INT NOT NULL,
    Age DECIMAL(4,1) NOT NULL
);

-- Boats Table
CREATE TABLE Boats (
    BoatID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Color VARCHAR(50) NOT NULL,
    Capacity INT NOT NULL
);

-- Reservations Table
CREATE TABLE Reservations (
    ReservationID INT PRIMARY KEY AUTO_INCREMENT,
    SailorID INT NOT NULL,
    BoatID INT NOT NULL,
    ReservationDate DATE NOT NULL,
    FOREIGN KEY (SailorID) REFERENCES Sailors(SailorID),
    FOREIGN KEY (BoatID) REFERENCES Boats(BoatID)
);
```

Sample Data

Sailors

SailorID	Name	Rating	Age
1	John Doe	5	25.5
2	Alice Smith	8	30.0
3	Bob Brown	7	22.3

Boats

BoatID	Name	Color	Capacity
1	Sea Breeze	Blue	6
2	Ocean Queen	White	10
3	Red Wave	Red	4

Reservations

ReservationID	SailorID	BoatID	ReservationDate
1	1	2	2025-01-01

ReservationID	SailorID	BoatID	ReservationDate
2	2	3	2025-01-05
3	3	1	2025-01-10

Practice Queries

1. **List all sailors who reserved the "Ocean Queen" boat:**
2. SELECT Sailors.Name
3. FROM Sailors
4. JOIN Reservations ON Sailors.SailorID = Reservations.SailorID
5. JOIN Boats ON Reservations.BoatID = Boats.BoatID
6. WHERE Boats.Name = 'Ocean Queen';
7. **Find the number of reservations made for each boat:**
8. SELECT Boats.Name, COUNT(*) AS ReservationCount
9. FROM Boats
10. JOIN Reservations ON Boats.BoatID = Reservations.BoatID
11. GROUP BY Boats.Name;
12. **Get sailors who have not made any reservations:**
13. SELECT Sailors.Name
14. FROM Sailors
15. LEFT JOIN Reservations ON Sailors.SailorID = Reservations.SailorID
16. WHERE Reservations.SailorID IS NULL;

This schema and queries offer a solid foundation to practice SQL concepts.

Sailors Database Schema

1. Sailors Table

Details about sailors, including their unique ID, name, rating, and age.

Column	Data Type	Constraints
SailorID	INT	Primary Key
Name	VARCHAR(100)	Not Null
Rating	INT	Not Null

Column	Data Type	Constraints
Age	DECIMAL(4,1)	Not Null

---

2. Boats Table

Details about boats, including their unique ID, name, color, and capacity.

Column	Data Type	Constraints
BoatID	INT	Primary Key
Name	VARCHAR(100)	Not Null
Color	VARCHAR(50)	Not Null
Capacity	INT	Not Null

3. Reservations Table

Tracks which sailor has reserved which boat on a specific date.

Column	Data Type	Constraints
ReservationID	INT	Primary Key
SailorID	INT	Foreign Key → Sailors
BoatID	INT	Foreign Key → Boats
ReservationDate	DATE	Not Null

SQL Implementation

```
-- Sailors Table
CREATE TABLE Sailors (
    SailorID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Rating INT NOT NULL,
    Age DECIMAL(4,1) NOT NULL
);

-- Boats Table
```

```
CREATE TABLE Boats (
    BoatID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Color VARCHAR(50) NOT NULL,
    Capacity INT NOT NULL
);

-- Reservations Table
CREATE TABLE Reservations (
    ReservationID INT PRIMARY KEY AUTO_INCREMENT,
    SailorID INT NOT NULL,
    BoatID INT NOT NULL,
    ReservationDate DATE NOT NULL,
    FOREIGN KEY (SailorID) REFERENCES Sailors(SailorID),
    FOREIGN KEY (BoatID) REFERENCES Boats(BoatID)
);
```

Sample Data

Sailors

SailorID	Name	Rating	Age
1	John Doe	5	25.5
2	Alice Smith	8	30.0
3	Bob Brown	7	22.3

Boats

BoatID	Name	Color	Capacity
1	Sea Breeze	Blue	6
2	Ocean Queen	White	10
3	Red Wave	Red	4

Reservations

ReservationID	SailorID	BoatID	ReservationDate
1	1	2	2025-01-01
2	2	3	2025-01-05
3	3	1	2025-01-10

Practice Queries

1. **List all sailors who reserved the "Ocean Queen" boat:**

```
SELECT Sailors.Name
FROM Sailors
JOIN Reservations ON Sailors.SailorID = Reservations.SailorID
JOIN Boats ON Reservations.BoatID = Boats.BoatID
WHERE Boats.Name = 'Ocean Queen';
```

2. **Find the number of reservations made for each boat:**

```
SELECT Boats.Name, COUNT(*) AS ReservationCount
FROM Boats
JOIN Reservations ON Boats.BoatID = Reservations.BoatID
GROUP BY Boats.Name;
```

3. **Get sailors who have not made any reservations:**

```
SELECT Sailors.Name
FROM Sailors
LEFT JOIN Reservations ON Sailors.SailorID = Reservations.SailorID
WHERE Reservations.SailorID IS NULL;
```

This schema and queries offer a solid foundation to practice SQL concepts.

Experiment-15

Suppliers and parts database

A **Suppliers and Parts Database** is another common relational database example, often used to demonstrate SQL concepts like joins, constraints, and aggregation.

Schema Design for Suppliers and Parts Database

Entities and Relationships:

1. **Suppliers**: Represents suppliers who provide parts, with details like ID, name, and location.
2. **Parts**: Represents parts supplied, with details like ID, name, and color.
3. **Supplies**: Represents the relationship between suppliers and parts, including the quantity supplied by a specific supplier.

Relational Schema:

Suppliers Table

- SupplierID (Primary Key): Unique identifier for each supplier.
- Name: Name of the supplier.
- Location: Location of the supplier.

Parts Table

- PartID (Primary Key): Unique identifier for each part.
- Name: Name of the part.
- Color: Color of the part.

Supplies Table

- SupplyID (Primary Key): Unique identifier for each supply record.
- SupplierID (Foreign Key): References SupplierID in the Suppliers table.
- PartID (Foreign Key): References PartID in the Parts table.
- Quantity: Quantity of the part supplied.

SQL Schema Implementation

```
-- Suppliers Table
CREATE TABLE Suppliers (
    SupplierID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Location VARCHAR(100) NOT NULL
);

-- Parts Table
CREATE TABLE Parts (
    PartID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Color VARCHAR(50) NOT NULL
);

-- Supplies Table
CREATE TABLE Supplies (
    SupplyID INT PRIMARY KEY AUTO_INCREMENT,
    SupplierID INT NOT NULL,
    PartID INT NOT NULL,
    Quantity INT NOT NULL,
    FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID),
    FOREIGN KEY (PartID) REFERENCES Parts(PartID)
);
```

Sample Data

Suppliers

SupplierID	Name	Location
1	ABC Corp	New York
2	XYZ Ltd	Los Angeles
3	Global Parts	Chicago

Parts

PartID	Name	Color
1	Bolt	Silver
2	Nut	Black
3	Screw	Gold

Supplies

SupplyID	SupplierID	PartID	Quantity
1	1	2	500
2	1	3	300
3	2	1	400
4	3	2	600

Practice Queries

1. **List all suppliers and the parts they supply:**

```
SELECT Suppliers.Name AS SupplierName, Parts.Name AS PartName, Supplies.Quantity
FROM Supplies
JOIN Suppliers ON Supplies.SupplierID = Suppliers.SupplierID
JOIN Parts ON Supplies.PartID = Parts.PartID;
```

2. **Find the total quantity supplied for each part:**

```
SELECT Parts.Name AS PartName, SUM(Supplies.Quantity) AS TotalQuantity
FROM Supplies
JOIN Parts ON Supplies.PartID = Parts.PartID
GROUP BY Parts.Name;
```

3. **Get the suppliers who supply the "Bolt":**

```
SELECT Suppliers.Name AS SupplierName
FROM Supplies
JOIN Suppliers ON Supplies.SupplierID = Suppliers.SupplierID
JOIN Parts ON Supplies.PartID = Parts.PartID
```

```
WHERE Parts.Name = 'Bolt';
```

4. **Find parts that are not supplied by any supplier:**

```
SELECT Parts.Name
FROM Parts
LEFT JOIN Supplies ON Parts.PartID = Supplies.PartID
WHERE Supplies.PartID IS NULL;
```

Normalization in the Database

1. **1NF**: Each table contains atomic values with no repeating groups.
2. **2NF**: No partial dependencies in the Supplies table (all non-primary key attributes depend on the full primary key).
3. **3NF**: No transitive dependencies, as attributes depend only on the primary key.

This schema offers a practical way to explore concepts in relational database design and query optimization.

VIVA QUESTIONS

Basic DBMS Questions

1. What is a database?
   A database is an organized collection of data stored electronically to facilitate access, management, and updating.
2. What is DBMS?
   A Database Management System (DBMS) is software used to store, retrieve, and manage data in databases.
3. What are the advantages of a DBMS?
   Data security, reduced redundancy, data integrity, concurrent access, and data consistency.
4. What is a schema?
   A schema is the logical structure of a database, including tables, relationships, and constraints.
5. What is the difference between DBMS and RDBMS?
   - DBMS: Manages data as files without relationships.
   - RDBMS: Manages data in tables with relationships.

Relational Database Concepts

6. What is a primary key?
   A unique identifier for each record in a table.
7. What is a foreign key?
   A field in one table that links to the primary key in another table.
8. What is a candidate key?
   A set of attributes that can uniquely identify a record in a table.
9. What is normalization?
   The process of organizing data to reduce redundancy and improve data integrity.
10. What are the different normal forms?
    - 1NF: Eliminates repeating groups.
    - 2NF: Removes partial dependency.
    - 3NF: Removes transitive dependency.
    - BCNF: Ensures every determinant is a candidate key.

11.**What is SQL?**
SQL (Structured Query Language) is used to interact with a database for querying, updating, and managing data.

12. **What are DDL commands?**
Data Definition Language commands: CREATE, ALTER, DROP.

13. **What are DML commands?**
Data Manipulation Language commands: INSERT, UPDATE, DELETE.

14. **What is the difference between DELETE and TRUNCATE?**

- DELETE: Removes rows with conditions; can be rolled back.
- TRUNCATE: Removes all rows; cannot be rolled back.

15. **What is the difference between HAVING and WHERE?**

- WHERE: Filters rows before grouping.
- HAVING: Filters groups after aggregation.

16.**What is a transaction?**
A sequence of database operations treated as a single logical unit of work.

17.**What are ACID properties?**

- **Atomicity**: All or nothing execution.
- **Consistency**: Data remains consistent.
- **Isolation**: Transactions do not interfere.

- **Durability**: Changes persist after a transaction.

18. **What is a deadlock?**
A situation where two or more transactions wait for each other indefinitely.

19. **What is the difference between COMMIT and ROLLBACK?**
    - COMMIT: Saves changes permanently.
    - ROLLBACK: Reverts changes.

20. **What is concurrency control?**
Mechanism to ensure multiple transactions execute without conflicts.

Indexing and Performance

21. **What is an index?**
A database structure to improve the speed of data retrieval operations.

22. **What is the difference between clustered and non-clustered indexes?**
    - **Clustered**: Sorts data physically.
    - **Non-clustered**: Does not affect data order; uses pointers.

23. **What is query optimization?**
Techniques to improve query performance by minimizing resource usage.

24. **What is the purpose of the EXPLAIN statement in SQL?**
It shows the execution plan of a query.

25. **What is a composite index?**
An index on multiple columns in a table.