

# **Master of Computer Applications (MCA)**

## **Database Management System Lab (DMCACO108P24)**

### **Self-Learning Material (SEM 1)**



**Jaipur National University  
Centre for Distance and Online Education**

---

**Established by Government of Rajasthan  
Approved by UGC under Sec 2(f) of UGC ACT 1956  
&  
NAAC A+ Accredited**

## **PREFACE**

In this guide, we will explore the fundamental concepts, principles, and best practices for effectively managing databases. Whether you are new to database management or looking to enhance your existing skills, this resource will provide valuable insights and guidance to help you successfully navigate the complex world of data storage and organization.

Databases are essential components of many applications and systems, and their proper design and management are crucial for ensuring efficiency, data integrity, and scalability. Through this preface, we invite you to embark on a journey of discovery and exploration into the world of databases.

By understanding the importance of database management, you will be equipped to make informed decisions, optimize database performance, ensure data integrity, and promote efficient data retrieval and analysis. Through this preface, we invite you to embark on a journey of discovery and mastery of database management principles that will empower you to harness the full potential of your data assets.

## TABLE OF CONTENTS

Sr. No.	Topic	Page No.
1.	Install Oracle RDBMS and create a Database.	1-2
2.	Familiarize yourself with Oracle SQL Developer or another Oracle-compatible SQL client.	2-3
3.	Create a schema for a university, including tables for students, courses, and instructors.	3-4
4.	Write SQL queries to insert data in to the tables you created.	4
5.	Write a query to retrieve very students from the student table.	5
6.	Create a database called "College" with two tables named "Students" and "Courses". Then, insert sample data into these tables and perform a simple join operation to retrieve student names along with the courses they are enrolled in.	5-6
7.	Write a query to find the courses by the highest and lowest number of registered students.	6-7
8.	Write a query to retrieve the average GPA of students in each course.	7
9.	Write a query to update a student GPA.	7-8
10.	Write a query to update a course's credit hours.	8
11.	Write a query to delete a student from the student table.	8
12.	Write a query to drop the course registration table	8
13.	Create a table for storing student addresses.	8-9
14.	Write SQL queries to insert Data in to the student addresses table.	9
15.	Write a query to retrieve a specific student's address by their student ID.	9
16.	Write a query to delete a student's address as of the student addresses table.	9
17.	Create a table for storing instructor's office hours.	9
18.	Write SQL queries to add data into the instruct or office hours table.	10
19.	Write a query to retrieve all instruct or office hours	10
20.	Write a query to retrieve a specific instructor's office hours by their employee ID	10
21.	Write a query to update an instructor's office hours	10
22.	Write a query to delete an instructor's office hours from the instructor office hours table.	10
23.	Create a table for storing course prerequisites.	10-11
24.	Consider a table named '\Employees' with the following columns: employee _id, first_ name, last_name, age, department, and salary. Write a"SQL query" to display the first name, last name, and salary of all employees working within the 'Finance' department.	11

25.	Consider a table named 'Students' by the follow columns: student_id, first_name, last_name, age, grade, and course_id. Write a SQL query to compute the average age of students in grade 10.	11
26.	Consider two tables named 'Orders' and 'Order_items'. The 'orders' table has the columns order_id, , and order_date, customer_id. The 'order_items' table has the column order_id, product_id, quantity, with price. Write a “SQL query” to find the total revenue generated on a specific date (e.g., '2023-03-31').	11
27.	Consider a table name 'products' by the following columns: product_id,product_name, category, and price. write down a SQL query to display the three most expensive products in each category	11-12
28.	Consider a table name 'customers' by the next columns: customer_id,first_name, last_name, Email, and Phone. Write down a SQL query to update the phone numbers of customers with the lastname 'Smith' by adding a '+1' prefix.	12
29.	Consider a table name 'books' with the next columns: book_id, title, author,genre, in addition to publication_year. Write down a SQL query to count the number of books published in each genre after 2010.	12
30.	Consider two tables name 'authors' and 'books'. The 'authors' table have the columns author_id, first_name, and last_name. The 'books' table has the columns book_id,title, author_id, and publication_year. Write down a SQL query to display the list of authors who have published at least three books.	12
31.	Consider a table name 'inventory' by the following columns: product_id,product_name, quantity, and price. Write down a “SQL query” to display the total value of the inventory(quantity*price)for each product with a value greater than 1000.	12-13
32.	Consider a table name 'events' with the following columns: event_id, event_name,start_date, end_date, and venue. Write down a SQL query to display the events scheduled to occur between '2023-04-02' and '2023-04-30', sorted by start date.	13
33.	Consider a table named 'users' with the following columns: user_Id, username, email,and registration_dAte. Write a SQL query to delete all users who registered more than two years ago(assuming the current date is '2023-03-31').	13
34.	Consider a table name 'sales' with the following columns: sale_id, product_id,sale_date, and quantity. Write down a “SQL query” to display total number of sales foreverymonthin2022.	13
35.	Consider two tables name 'students' and 'enrollments'. The 'students' table has thecolumns student_id, first_name, and	13

	last_name	
36.	Consider a table name 'orders' with the following columns: order_id, customer_id, order_date, and total_amount. Write down a “SQL query” to find the total revenue generated per month in 2022..	13-14
37.	Consider a table name 'employees' by the following columns: employee_id, first_name, last_name, hire_date, and salary. Write down a SQL query to display the employees hired with in the last 6 months (assuming the current date is '2023-03-31').	14
38.	Consider a table name 'cities' with the following columns: city_id, city_name, country, and population. Write down a SQL query to display the top 5 most populous cities in ascending order.	14
39.	Consider a table name 'employees' with the following columns: employee_id, first_name, last_name, department, and salary. Write down a SQL query to find the employees with the highest salary in each department.	14-15
40.	Consider two tables named 'students' and 'courses'.	15
41.	Write a query to locate the courses offered in a specific semester.	15
42.	Write a query to find the instructors teaching a specific course	15-16
43.	Write a query to locate the students by the highest GPA in a specific course.	16
44.	Write a query to find courses by means of no registered students	16
45.	Write a Query to locate the top 4 students with the highest GPA.	16
46.	Write a query to find the Top 5 courses with the maximum average GPA	16
47.	Write a query to find the Top 5 instructors with the Highest average student GPA	17
48.	Create a view to displays the student ID, name, and total credit hours of the courses they are registered for	17
49.	Create a view to displays the instructor ID, name, and total credit hours of the courses they are teaching	17
50.	Create a stored procedure to enroll a student in a course	18
51.	Create a store procedure to drop a course used for a student.	18
52.	Create a store procedure to add a fresh course to the course table.	18-19
53.	Create a store procedure to delete a course from the course table	19
54.	Create a store procedure to add a new instructor to the instructor table	19
55.	Create a stored procedure to delete an instructor from the instructor table.	19
56.	Create a store procedure to revise a student's GPA.	20
57.	Create a store procedure to bring up to date a course's credit hours	20
58.	Create a stored procedure to up date an instructor's office hours.	21
59.	Create a function to compute the average GPA of students in a specific course.	21

60.	Create a function to compute the total credit hours earned by a specific student..	22
61.	Create a function to calculate the total credit hours taught by a specific instructor.	22-23
62.	Create a trigger to update the total credit hours earned by a student when they enroll or drop a course	23-24
63.	Create a trigger to update the total credit hours earned by a student when they enroll or drop a course	24
64.	Create a table for storing user login logs.	24
65.	Write SQL queries to add data into the user login logs	24
66.	Write a query to retrieve a specific user's login logs By their user ID	24
67.	Write a query to delete a specific user's login logs from the user login log stable	25
68.	Create An Index on the student table to improve query performance.	25
69.	Create an Index on the course table to improve query performance.	25
70.	Analyze the performance of aspecific query using the Oracle Explain Plan feature.	25
71.	Use SQL subqueries to find students who are registered for the same course as a specific student	25
72.	Use SQL subqueries to find instructors who have taught the same course as a specific instructor.	26
73.	Write a query to implement pagination for a large result set, such as a list of all students.	26
74.	Write aquery to implement full-text search for student names or course titles	26
75.	Use SQL transactions to ensure data integrity when enrolling students in courses or updating their GPA.	26-27
76.	Optimize a slow-running query using indexes, subqueries, or other query optimization techniques.	27-28
77.	Use SQL join store cover a list of students and the course they are registered for	28
78.	Use SQL join store cover a record of instructors and the course they are teaching.	28
79.	Use SQL join store cover a list of courses and their prerequisites	28
80.	Write down a SQL query to create a normalized schema for an online store, including tables for customers, products, orders, and order items. Ensure each table has appropriate primary and foreign keys.	28-29
81.	Write down a “SQL query “ to create an index On the product name in the "Products" table, assuming the product name column is called "product_name	29
82.	Write a “SQL query” to find the top 3 Best-selling products in the	29

	online store, based on the total quantity sold.	
83.	Write a “SQL query” to find the total revenue generate by the online store for a given date range, using the "Orders" and "Order_ Items" tables.	30
84.	Create a store procedure to insert a new customer record into the "Customers" table, checking for duplicate email addresses and returning an appropriate error message if a duplicate is found	30-31
85.	Create a store procedure to update the stock level meant for a product in the "Products" table, considering the stock level should not be less than zero.	31-32
86.	Write a SQL query to implement a transaction that inserts a new order and its associated order items into the "Orders" and "Order_Items" tables, ensuring dataconsistency.	31-32
87.	Write a SQL query to Create a view that displays the whole revenue generated by each customer.	32
88.	Implement a trigger that automatically calculates and update the total price of an order in the "Orders" table when a new record is inserted into the "Order Items"table.	33
89.	Write a SQL query to create a schema for a many-to-many relationship among students and courses, using a junction table for enrollments.	33
90.	Write a SQL query to uncover the average grade of all students in a specific course, using the "Students", "Courses", and "Enrollments" tables.	34
91.	Write down a SQL query to identify clients who have placed more than 5 orders in the online store, using the "Customers" and "Orders" tables.	34
92.	Write a SQL query to find the products that have not been ordered in the last 30 days, using the "Products", "Orders", and "Order_ Items" tables.	34
93.	Implement a trigger that checks the stock point of a product before inserting a new record into the "Order_Items" table. If the stock level is insufficient, cancel the insertion and return an error message.	34
94.	Create a stored procedure to calculate and update the total price of an order after adding, updating, or deleting an order item.	35
95.	Write a SQL query to Create an index on the "Orders" table to optimize the search for orders placed within a specific date range.	35
96.	Write a “SQL query” to find the total number of enrollments for each course, sorted by the number of enrollments in descending order.	35
97.	Create a view to displays the number of products sold in each product category, using the "Products" and "Order_Items "tables.	36

98.	Implement a Foreign key constraint on the "Enrollments" table to ensure that a record can only be added if the corresponding student and course IDs exist in the "Students" and "Courses" tables.	37
99.	Write a SQL query to identify the top three customers who have spent the most money on top of the online store, based on the total price of their orders.	37
100.	Write down a SQL query to find the students who have taken at least 1 course from each department, using the "Students", "Courses", "Enrollments", and "Departments" tables. Assume there is a "department_id" column in both the "Courses" and "Departments" tables.	37-38



## 1. Install Oracle RDBMS and create a Database.

Installing “Oracle RDBMS” and creating a Database is a multi-step process that can be broken down into the following steps:

- Download Oracle Database Software: Go to the Oracle Database download page(<https://www.oracle.com/database/technologies/>) and choose the version you want to install (e.g., Oracle Database 19c). Download the proper installer for your operating system (Linux, Windows, or macOS).
- Install the Oracle Database Software: Follow the setting up steps specific to your Operating system. For detailed instructions, consult Database of Oracle Installation Guide (<https://docs.oracle.com/en/database/oracle/oracle-database/index.html>) for your chosen version and operating system.
- create an Oracle Database: After installing “Oracle Database” software, you can create a

### “Database using the Database Configuration Assistant” (DBCA). Follow these steps:

a. Open the database Configuration Assistant:

- On Windows, click "Start," search for "Database Configuration Assistant" and click on it.
- On Linux, open a terminal and enter **dbca** to launch the Database Configuration Assistant.

b. Choose "Create a Database" and click "Next."

Choose the appropriate template for your database (e.g., General Purpose or Transaction Processing) and click "Next."

Enter a Global Database Name and System Identifier “SID” for your database. The Global Database Name should be in the format "database\_name.domain\_name" (e.g., "mydb.example.com"). The SID is a unique identifier for your database instance. Click "Next."

c. Choose the storage options for your database, such as file system or Automatic Storage Management (ASM). Click "Next."

d. Set the memory allocation for your database by choosing either automatic or custom memory management. Click "Next."

e. Choose the appropriate character set for your database and click "Next."

f. Configure security settings for your database, such as enabling Transparent Data Encryption (TDE) and setting a password for the SYS and SYSTEM accounts. Click "Next."

g. Configure the management options for your database, such as enabling Oracle Enterprise Manager Database Express and setting a password for the DBSNMP account. Click "Next."

C

h. R  
Review the summary of your database configuration and click "Finish" to create the database.

- **join to the Oracle Database:** Use SQL\*Plus, SQL Developer, or another database management tool on the way to connect to your newly created Oracle Database using the connection details (SID, username, and password) you provided during the database creation process.

Now you have installed Oracle RDBMS and created a database. You can start creating tables, inserting data, and querying the database as needed

## 2. Familiarize yourself with Oracle SQL Developer or another Oracle-compatible SQL client.

“Oracle SQL” Developer is a popular, free “Integrated Development Environment” (IDE) that simplifies working with Oracle databases. It provides a powerful and intuitive interface for managing database objects, running “SQL queries”, and developing “PL/SQL code”.

Here's how to get started with “Oracle SQL Developer”:

1. Download “Oracle SQL” Developer: Go to the Oracle SQL Developer download page (<https://www.oracle.com/tools/downloads/sqldev-downloads.html>) and download the appropriate version for your operating system (Windows, macOS, or Linux).
2. Installing Oracle SQL Developer: Follow the installation instructions for your operating system provided in the Oracle SQL Developer documentation (<https://docs.oracle.com/en/database/oracle/sql-developer/index.html>).
3. Launch Oracle SQL Developer: Start Oracle SQL Developer by running the executable file (sqldeveloper.exe on Windows or sqldeveloper.sh on Linux/macOS) located in the installation folder.
4. Create a Database Connection: To connect to your Oracle Database, you need to set up a fresh database connection. Click the "+" icon within the "Connections" tab in the left pane to open the "NewConnection" dialog.

Fill in the required details, such as:

- **ConnectionName:** A Unique Name For The Connection.
- **Username:** The database user account (e.g., SYSTEM or another user account you've created).
- **Password:** The Password For The User Account.

- Hostname: The hostname or "IP address" of the server host your Oracle Database.
- Port : The listener port for your Oracle Database (default is 1521).
- SID or Service Name: The SID or Service Name of your Oracle Database.

Click "Test" to ensure the connection settings are correct, then click "Connect" to establish a connection to the database.

5. Explore Oracle SQL Developer Features: With Oracle SQL Developer, you can manage your database, develop and debug PL/SQL code, run SQL queries, and more. Familiarize yourself with the following features:

- SQL Worksheet: Write, execute, and save SQL queries, PL/SQL code, and scripts. Access it by right-clicking a connection and selecting "Open SQL Worksheet" or clicking the "SQL Worksheet" button toolbar.
- Object Browser: Explore and manage database objects (tables, indexes, views, etc.) in the "Connections" tab. You can create, edit, and delete objects by right-clicking selecting the appropriate options.
- Data Import and Export: Import data from external files (CSV, Excel, XML, etc.) or export data from tables and views to various file formats. Access these options by right-clicking a table or view and selecting "Import Data" or "Export Data."
- PL/SQL Debugging: Debug PL/SQL code by setting breakpoints, stepping through code, and examining variable values. Open a PL/SQL object (procedure, function, package, etc.) in the editor, set breakpoints, and click the "Debug" button on the toolbar to start a debugging session.

### 3. Create a schema for a university, including tables for students, courses, and instructors.

```
-- Creating table for students CREATE TABLE students (
```

```
Student_id NUMBER PRIMARY KEY,
```

```
First_name VARCHAR2(50), Last_name VARCHAR2(50), Birth_date DATE,
```

```
major VARCHAR2(50)
```

```
);
```

```
-- Creating table for courses CREATE TABLE courses (
```

```

course_id NUMBER PRIMARY KEY, course_name VARCHAR2(100),
course_description VARCHAR2(1000), instructor_id NUMBER);

-- Creating table for instructors CREATE TABLE instructors (instructor_id NUMBER
PRIMARY KEY, first_name VARCHAR2(50), last_name VARCHAR2(50),
department VARCHAR2(50)
);

-- Add foreign key constraint on courses referencing instructors ALTER TABLE
courses
ADD CONSTRAINT fk_instructor FOREIGN KEY
(instructor_id) REFERENCES instructors (instructor_id);

```

#### 4. Write SQL queries insert data into the tables created.

```

--Inserting data into instructors table
INSERT INTO instructors (instructor_id, first_name, last_name,
department) VALUES (1, 'John', 'Doe', 'Computer Science');

--Inserting data into student table
INSERT INTO students (student_id, first_name, last_name, birth_date,
major) VALUES (1, 'Jane', 'Smith', TO_DATE('1998-05-17', 'YYYY-MM-
DD'), 'Computer
Science');

--Inserting data into courses table

```

**5. Write a query to retrieve every student from the student table.**

```
SELECT * FROM students;
```

**Output:**

```
STUDENT_ID|FIRST_NAME|LAST_NAME| BIRTH_DATE|MAJOR
```

```
-----  
1         |Jane       |Smith   | 17-MAY-98| Computer Science
```

**6. Create a database called "College" with two tables named "Students" and "Courses". Then, insert sample data into these tables and perform a simple join operation to retrieve student names along with the courses they are enrolled in.**

**• Creating the "College" database:**

```
CREATE DATABASE College;
```

**• Create the "Students" table:**

```
USE College;
```

```
CREATE TABLE Students
```

```
(Student_id INT PRIMARY
```

```
KEY,
```

```
student_name VARCHAR(50)
```

```
NOT NULL, Course_id INT
```

```
);
```

**• Create the "Courses" table:**

```
CREATE TABLE
```

```
Courses
```

```
(Course_id INT PRIMARY
```

```
KEY,
```

```
course_name VARCHAR(50) NOT NULL
```

```
);
```

**• Insert sample data into the "Students" table:**

```
INSERT INTO students (student_id, student_name,
```

```
Course_id) VALUES (1, 'Alice', 101),
```

```
(2,
```

```
'Bob', 102),
```

(3,'Charlie',101);

- **insert sample data into the "Courses" table:** INSERT INTO Courses (course\_id, course\_name) VALUES(101, 'Mathematics'),

(102,'Physics');

- **Performing a simple join operation to retrieve student names all along with the courses they are enrolled in:**

```
SELECT      student_name and
course_name FROM Students
JOIN Courses ON Students.Course_id with Courses.course_id;
```

**7. Write a query to find the courses by the highest and lowest number of registered students.**

```
WITH Course Counts AS(
  SELECT   Course_id,   COUNT(student_id)   AS
  num_students
  FROM Students
  GROUP BY Course_id
)
, MinMaxCounts AS(
  SELECT   MIN(num_students) AS min_students,   MAX(num_students) AS
  max_students FROM Course Counts
)
SELECT      Courses.course_id,           Courses.course_name,
CourseCounts.num_students FROM Courses
JOIN CourseCounts ON Courses.course_id=CourseCounts.course_id
JOIN      MinMaxCounts      ON      CourseCounts.num_students      =
      MinMaxCounts.min_students      OR
      CourseCounts.num_students=MinMaxCounts.max_students;
```

This Query Consists Of Three Parts:

1. The **CourseCounts** Common Table Expression (CTE) calculate the number of registered student for each course by grouping the **Students** table by **course\_id** and counting the **student\_ids**.
2. The **MinMaxCounts** CTE finds the minimum and maximum number of registered students among all courses by selecting the **MIN** and **MAX** of the **num\_students** column from the **Course Counts** CTE.
3. The key query joins the **Courses**, **CourseCounts**, and **MinMaxCounts** tables to find and display the course ID, course name, and amount of registered students for the courses with the highest and lowest number of registered students.

### 8. Write a query to retrieve the average GPA of students in each course.

```
SELECT Courses.course_id, Courses.course_name, AVG(Students.GPA) AS
average_gpa FROM Students
JOIN Courses ON Students.course_id with Courses.course_id
GROUPBY Courses.course_id,Courses.course_name;
```

This query performs the following operations:

1. join the **Students** and **Course** stables on the **course\_id** column.
2. group the joined records by **course\_id** and **course\_name**(from the **Course**stable).
3. Calculates the Average GPA students in each group using the AVG() function.

The result of this query will display the course ID, course name, and average GPA of students in each course.

### 9. Write a query to update a student GPA.

To revise a student's GPA, you would first need to know the structure of your database, particularly the name of the table that holds the student information and the names of the columns for the student ID and GPA. Assuming the table name is "students" and the columns are "student\_id" and "gpa", you could write a query like this:

```
UPDATE
students SET gpa = new_gpa
WHERE student_id = target_student_id;
```

Replace `new_gpa` with the updated GPA value (e.g., 3.5) and `target_student_id` with the ID of the student whose GPA you desire to update (e.g., 12345). Your final query would look like this:

```
UPDATE
studentsSETgpa=
3.5
WHEREstudent_id=12345;
```

Before running the query, make sure to replace the table and column names if they are different in your database.

### **10. Write A query to update a course's credit hours.**

To update a course's credit hours, you would need to know the structure of your database, specifically the name of the table holding the course information and the names of the columns for the course ID and credit hours. Assuming the table name is "courses" and the columns are "course\_id" and "credit\_hours", you could write a query like this:

```
UPDATE courses
SET credit_hours = new_credit_hours WHERE course_id =
target_course_id;
```

Replace `new_credit_hours` with the updated credit hours value (e.g., 4) and `target_course_id` with the ID of the course whose credit hours you want to update (e.g., 'CSCI101'). Your final query would look like this:

```
UPDATEcoursesSE
Tcredit_hours=4
WHERE course_id = 'CSCI101';
```

### **11. Write a query to delete a student from the student table.**

```
DELETE FROM student
WHEREstudent_id=<student_id_to_delete>;
```

### **12. Write a query to drop the course registration table.**

```
DROPTABLEcourse_registration;
```

### **13. Create a table for storing student addresses.**



```
CREATE TABLE student_addresses(address_id SERIAL PRIMARY KEY,
    Student_id INT REFERENCES student (student_id), street VARCHAR(250),
    city VARCHAR(255), state VARCHAR(255),
    postal_code VARCHAR(255), country VARCHAR(255)
);
```

**14. Write SQL queries to insert data into the student addresses table.**

```
INSERT INTO student_addresses (student_id, street, city, state, postal_code,
country) VALUES (<student_id>, <street>, <city>,
<state>, <postal_code>, <country>);
```

**15. Write a query to retrieve a specific student's address by their student ID.**

```
SELECT * FROM student_addresses
WHERE student_id = <Student_id_to_search>;
```

**16. Write a query to delete a student's address as of the student address table.**

```
DELETE FROM student_addresses
WHERE student_id = 1; -- Replace 1 with the desired student ID
```

**17. Create a table for storing instructor's office hours.**

```
CREATE TABLE instructor_office_hours (id SERIAL PRIMARY KEY,
    instructor_id INT NOT NULL, day_of_week VARCHAR(15) NOT
    NULL, start_time TIME NOT NULL,
    End_time TIME NOT NULL
```

**18. Write SQL queries add data into the instructor office hours table.**

```
INSERT INTO instructor_office_hours (instructor_id, Day_of_week, start_time, end_time) VALUES (1,'Monday','10:00:00', '11:00:00');
```

```
INSERT INTO instructor_office_hours (instructor_id, Day_of_week, start_time, end_time)VALUES (2, 'Tuesday', '14:00:00', '16:00:00');
```

```
INSERT INTO instructor_office_hours (instructor_id, Day_of_week, start_time, end_time)VALUES(1,'Thursday','10:00:00', '12:00:00');
```

**19. Write a query to retrieve all instructor office hours.**

```
SELECT*FROMinstructor_office_hours;
```

**20. Write a query to retrieve a specific instructor's office hours their employeeID.**

```
SELECT*FROMinstructor_office_hours  
WHEREinstructor_id=1;--Replace1 with the desired instructor ID
```

**21. Write A Query To Update An Instructor's Office Hours.**

```
UPDATEinstructor_office_hours  
SETStart_time= '11:00:00',end_time='13:00:00'  
WHERE id=1;--Replace1 with thedesiredofficehours record ID
```

**22. Write a query to delete an instructor's office hours from the instructor office hours table.**

```
DELETEFROM instructor_office_hours  
WHERE id=1;--Replace1 with the desired office hours record ID
```

**23. Create a table for storing course prerequisites.**

```
CREATE TABLE Course_prerequisites(id SERIAL PRIMARYKEY,  
course_id INT NOT NULL, prerequisite_id INT NOT NULL  
);
```

- 24. Consider a table named 'Employees' with the following columns: employee\_id, first\_name, last\_name, age, department, and salary. Write a "SQL query" to display the first name, last name, and salary of all employees working within the 'Finance Department.'**

```
SELECT First_name ,last_name,  
salaryFROMEmployees  
WHERE  
Department='Finance';
```

- 25. Consider a table named 'Students' by the follow columns: student\_id, first\_name, last\_name, age, grade, and course\_id. Write a SQL query to compute the average age of students grade 10.**

```
SELECT AVG(age) AS average_age FROM Students  
WHERE grade=10;
```

- 26. Consider two tables named 'Orders' and 'Order\_items'. The 'orders' table has the columns order\_id, , and order\_date, customer\_id. The 'order\_items' table has the column order\_id, product\_id, quantity, with price. Write a "SQL query" to find the total revenue generated on a specific date (e.g., '2023-03-31').**

```
SELECT SUM(quantity * price) AS total_revenue FROM orders  
JOIN orders.order_id ON order_items = order_id.order_item WHERE order_date  
='2023-03-31';
```

- 27. Consider a table name 'products' by the following columns: product\_id, product\_name, category, and price. write down a SQL query to display the three most expensive products in each category.**

```
SELECT p1.product_id, p1.product_name, p1.category,  
p1.price FROM productsp1
```

```

WHERE (SELECT COUNT(*)
FROM productsp2
WHERE p2.category= P1.category ANDp2.price> P1.price
)< 3
ORDERBY p1.category,p1.priceDESC;

```

**28. Consider a table name 'customers' by the next columns: customer\_id,first\_name, last\_name, Email, and Phone. Write down a SQL query to update the phone numbers of customers with the last name 'Smith' by adding a '+1' prefix.**

```

UPDATE customers
SET phone = CONCAT('+1', phone) WHERE last_name = 'Smith';

```

**29. Consider a table name 'books' with the next columns: book\_id, title, author,genre, in addition to publication\_year. Write down a SQL query to count the number of books published in each genre after 2010.**

```

SELECT genre, COUNT(*) AS book_count FROM books
WHERE publication_year> 2010 GROUPBYgenre;

```

**30. Consider two tables name 'authors' and 'books'. The 'authors' table have the columnsauthor\_id, first\_name, and last\_name. The 'books' table has the columns book\_id,title, author\_id, and publication\_year. Write down a SQL query to display the list of authors who have published at least three books.**

```

“SELECT a.author_id, a.first_name, a.last_name, COUNT(b.book_id) AS
book_count FROM authors a
JOIN books b ON a.author_id = b.author_idGROUP BY a.author_id, a.first_name,
a.last_nameHAVINGCOUNT(b.book_id) >=3;”

```

**31. Consider a table name 'inventory' by the following columns: product\_id,product\_name, quantity, and price. Write down a “SQL query” to display the total value of the inventory(quantity\*price)for each product with a value greaterthan1000.**

```

SELECTproduct_id, product_name, quantity, price, (quantity*price) AS inventory_value

```

```
FROM inventory
WHERE (quantity*price)>1000;
```

- 32. Consider a table name 'events' with the following columns: event\_id, event\_name, start\_date, end\_date, and venue. Write down a SQL query to display the events scheduled to occur between '2023-04-02' and '2023-04-30', sorted by start\_date.**

```
SELECT event_id, event_name, start_date, end_date, venue FROM events
WHERE start_date BETWEEN '2023-04-02' AND '2023-04-30'
ORDER BY start_date;
```

- 33. Consider a table named 'users' with the following columns: user\_id, username, email, and registration\_date. Write a SQL query to delete all users who registered more than two years ago (assuming the current date is '2023-03-31').**

```
DELETE FROM users
WHERE registration_date < DATE_SUB ('2023-03-31', INTERVAL 2 YEAR);
```

- 34. Consider a table name 'sales' with the following columns: sale\_id, product\_id, sale\_date, and quantity. Write down a "SQL query" to display the total number of sales for every month in 2022.**

```
"SELECT YEAR ( sale_date) AS sale_y
ear , MONTH(sale_date) AS sale_month, COUNT(*) AS sale_count
FROM sales
WHERE YEAR(sale_date) = 2022 GROUP BY sale_year, sale_month;"
```

- 35. Consider two tables named 'students' and 'enrollments'. The 'students' table has the columns student\_id, first\_name, and last\_name. The 'enrollments' table has the columns enrollment\_id, student\_id, course\_id, and semester. Write down a SQL query to display the list of students who are not enrolled in a few courses for the 'Spring 2023' semester.**

```
"SELECT s.student_id, s.first_name,
s.last_name FROM students
LEFT JOIN enrollments e ON s.student_id = e.student_id AND e.semester = 'Spring
2023 WHERE e.enrollment_id IS NULL;"
```

- 36. Consider a table name 'orders' with the following columns: order\_id, customer\_id, order\_date, and total\_amount. Write down a "SQL query" to find the total revenue generated per month in 2022.**

```
“SELECT YEAR(oRder_date) AS order_year, MONTH(oRder_date) AS
order_month,SUM(total_amount) AS monthly_revenue
FROMoRders
WHERE YEAR(order_date) = 2022 GROUPLY order_year, order_month;”
```

- 37. Consider a table name 'employees' by the following columns: employee\_id, first\_name, last\_name, hire\_date, and salary. Write down a SQL query to display the employees hired within the last 6 months(assuming the current date is '2023-03-31').**

```
“SELECT employee_id, first_name, last_name, hire_date, salary FROM employees
WHERE hire_date>DATE_SUB('2023-03-31', INTERVAL 6 MONTH);”
```

- 38. Consider a table named 'cities' with the following columns: city\_id, city\_name,country, and population. Write down a SQL query to display the top 5 most populous cities in ascending order.**

```
SELECT city_id, city_name, country, population FROM cities
ORDER BY population DESC LIMIT 5;
```

- 39. Consider a table name 'employees' with the following columns: employee\_id, first\_name, last\_name, department, and salary. Write down a SQL query to find the employees with the highest salary in each department.**

Step1:

First, we need to find the highest salary for each department. To do this, we use the GROUP BY clause to group the records in the department and the MAX() function to get the greatest salary in each group.

```
SELECT department, MAX(salary) AS highest_salary FROM employees
GROUP BY department;
```

Step2:

Now that we have the highest salary for each department, we need to join the result of the previous query with the original 'employees' table to get the employee details.

```
SELECT e.employee_id, e.department, e.first_name, e.salary FROM employees
,e.last_name JOIN (
SELECT department, MAX (Salary) AShighest_Salary
```

```
FROM employees GROUP BY department
)ON e.department=d.department AND e.salary=d.highest_salary;
```

The inner query (subquery) calculates the highest salary designed for each department, and the outer query joins the 'employees' table with the result of the subquery to get the employee details.

- 40. Consider two tables named 'students' and 'courses'. The 'students' table has the columns first\_name, student\_id, and last\_name . The 'courses' table has the columns course\_id, course\_name, with instructor. Write a SQL query to find the students who have not taken any courses taught by a specific instructor (e.g.,'JohnSmith').**

Step 1:

Filter the 'courses' table to get the courses taught by the specific instructor.  

```
SELECT course_id
FROM courses
WHERE instructor='JohnSmith';
```

Step2:

Join the 'students' table by the 'courses' table using a LEFT JOIN to get the list of students who have taken courses taught by the specific instructor. Filter the result to include only students who haven't taken any of the instructor's courses.

```
SELECT DISTINCT s.student_id, s.first_name, s.last_name FROM students
LEFT JOIN courses c ON s.course_id = c.course_id AND c.instructor = 'John
Smith' WHERE c.course_id IS NULL;
```

The LEFT JOIN ensures that all students are included in the result, even if they haven't taken any courses taught by the specific instructor. The DISTINCT keyword is used to remove duplicate entries in case a student is enrolled in multiple courses not taught by the instructor.

- 41. Write a query to locate the courses offered in a specific semester.**

```
SELECT *FROM
coursesWHERE

semester='Fall2023';
```

- 42. Write a query to find the instructor's teaching specific course.**

```
“SELECT i.instructor_id ,i.instructor_name FROM instructors JOIN course_instructors.ci ON
i.instructor_id = ci.instructor_id WHERE ci.course_id = 'CS101' ;”
```

**43. Write a query to locate the students by the highest GPA in a specific course.**

```
SELECT s.student_id, s.student_name, s.gpa FROM students s JOIN course_registrations cr ON
s.student_id = cr.student_id WHERE cr.course_id = 'CS101' AND s.gpa = (
SELECT MAX(gpa) FROM students st
JOIN course_registrations crt ON st.student_id = crt.student_id WHERE crt.course_id =
'CS101'
);
```

**44. Write a query to find courses by means of non registered students.**

```
“SELECT c.course_id, c.course_name FROM courses
LEFT JOIN course_registrations cr ON c.course_id = cr.course_id WHERE cr.student_id IS
NULL;”
```

**45. Write a Query to locate the top 4 students with the highest GPA.**

```
SELECT id, name, GPA FROM students ORDER BY GPA DESC LIMIT 4 ;
```

**46. Write a query to find the Top 5 course with the maximum average GPA.**

```
“SELECT c.id, c.name, AVG(r.grade) as average_gpa FROM courses c
JOIN registrations r ON c.id = r.course_id GROUP BY c.id, c.name
ORDER BY average_gpa DESC
LIMIT
```

5;”



**47. Write a query to find the Top 5 instructors with the Highest average student GPA.**

```
SELECT i.id, i.name, AVG(r.grade) as
average_gpa FROM instructors i
JOIN course_instructors ci ON i.id =
ci.instructor_id JOIN registrations r ON
ci.course_id = r.course_id GROUP BY i.id, i.name
ORDER BY average_gpa
DESC LIMIT 5;
```

**48. Create a view to display the student ID, name, and total credit hours of the course they are registered for.**

```
CREATE VIEW student_credit_hours AS
SELECT s.id as student_id, s.name as student_name, SUM(c.credit_hours)
as total_credit_hours
FROM student s
JOIN registrations r ON s.id =
r.student_id JOIN courses c ON
r.course_id = c.id GROUP BY s.id, s.name;
```

**49. Create a view to display the instructor ID, name, and total credit hours of the courses they are teaching.**

```
CREATE VIEW instructor_credit_hours AS
SELECT i.id as instructor_id, i.name as instructor_name, SUM(c.credit_hours)
as total_credit_hours
FROM instructors i
JOIN course_instructors ci ON i.id =
ci.instructor_id JOIN courses c ON ci.course_id = c.id
GROUP BY i.id, i.name;
```

**50. Create a stored procedure to enroll a student in a course.**

```
CREATE PROCEDURE
EnrollStudent@StudentIDINT,
@CourseID
INTAS
BEGIN
INSERT INTO Enrollment (StudentID,
CourseID)VALUES(@StudentID,@CourseID)
;
END
;GO
```

**51. Create a stored procedure to drop a course used for a student.**

```
CREATE PROCEDURE
DropCourse@studentIDINT,
@courseID
INTAS
BEGIN
DELETEFROM Enrollment
WHERE StudentID = @StudentID AND courseID =
@courseID;END;
GO
```

**52. Create a stored procedure to add a fresh course to the course table.**

```
CREATE PROCEDURE AddCourse
@CourseID INT,
@CourseNameNVARCHAR(25
5),@CreditHoursINT
ASBE
GIN
```

```
INSERT INTO Courses (CourseID, CourseName,
CreditHours)VALUES(@CourseID,@CourseName,@CreditHours);
END;GO
```

**53. Create a stored procedure to delete courses from the course table.**

```
CREATE PROCEDURE DeleteCourse@courseID INT
ASBEGIN
DELETFROMCourses
WHERE CourseID = @CourseID;END;
GO
```

**54. Create a stored procedure to add an instructor to the instructor table.**

```
CREATE PROCEDURE
AddInstructor@InstructorIDINT,@InstructorNameNVARCHAR(255),@OfficeHoursNV
ARCHAR(255)
ASBEGIN
INSERT INTO Instructors (InstructorID, InstructorName,
OfficeHours)VALUES(@InstructorID,@InstructorName,@OfficeHours);
END;GO
```

**55. Create a stored procedure to delete instructors from the instructor table.**

```
CREATE PROCEDURE DeleteInstructor
@InstructorID INTAS
BEGIN
DELETFROMInstructors
WHERE InstructorID = @InstructorID;END;
GO
```

**56. Create a stored procedure to revise a student's GPA.**

```
CREATE PROCEDURE UpdateStudentGPA@StudentIDINT,  
@NewGPADECIMAL(4, 2)AS  
BEGIN  
UPDATEStudents  
SET GPA=@NewGPA  
WHERE StudentID = @StudentID;END;  
GO
```

**57. Create a stored procedure to bring up to date a course's credit hours.**

```
CREATE PROCEDURE UpdateCourseCreditHours@CourseID INT,  
@NewCreditHours INTAS  
BEGIN  
UPDATECourses  
SET CreditHours = @NewCreditHoursWHERECourseID=@CourseID;
```

```
END
;GO
```

**58. Create a stored procedure to update an instructor's office hours.**

```
CREATE PROCEDURE
UpdateInstructorOfficeHours @InstructorID INT,
@NewOfficeHours NVARCHAR(2
55) AS
BEGIN
UPDATE Instructors
SET OfficeHours =
@NewOfficeHours WHERE
InstructorID = @InstructorID; END;
GO
```

**59. Create a function to compute the average GPA of students in a specific course.**

```
CREATE FUNCTION AvgGPAByCourse
(@CourseID INT) RETURNS DECIMAL(4,2)
AS BE
GIN
RETURN (
SELECT AVG(GPA)
FROM Students
JOIN Enrollment ON Students.StudentID =
Enrollment.StudentID WHERE Enrollment.CourseID = @Course
ID
); EN
D; G
O
```

### 60. Create a function to compute the total credit hours earned by a specific student.

Assuming you have a table enrollments by columns student\_id, course\_id, and credit\_hours, the function to calculate the total credit hours earned by a specific student can be created as follows:

```
CREATE FUNCTION total_credit_hours_student(student_id INT) RETURNS INT
AS $$
DECLARE
    total_hours
INT;
BEGIN
    SELECT SUM(credit_hours) INTO total_hours FROM enrollments WHERE student_id =
    $1;
    RETURN
total_hours;
END;
$$ LANGUAGE plpgsql;
```

### 61. Create a function to calculate the total credit hours taught by a specific instructor.

To calculate the whole credit hours taught by a specific instructor, you can create a similar function. Assuming you have a table courses with columns instructor\_id and credit\_hours:

```
CREATE FUNCTION total_credit_hours_instructor(instructor_id INT) RETURNS INT
AS $$
DECLARE
    TOTAL_hours
INT;
BEGIN
    SELECT SUM(credit_hours) INTO total_hours FROM courses WHERE instructor_id
    = $1;
    RETURN total_hours;
END;
$$ LANGUAGE plpgsql;
```

### 62. Create a trigger to update the total credit hours earned by a student when they enroll or drop a course.

Assuming you have a table students with columns id and total\_credit\_hours, create a trigger to update the total credit hours earned by a student when they enroll or drop a course:

```

CREATE OR REPLACE FUNCTION update_student_credit_hours() RETURNS TRIGGER AS
$$BE
GIN
  IF(TG_OP='INSERT') THEN
    UPDATE students SET total_credit_hours = total_credit_hours +
NEW.credit_hours WHERE id=NEW.student_id;
  ELSIF (TG_OP = 'DELETE') THEN
    UPDATE students SET total_credit_hours = total_credit_hours -
OLD.credit_hours WHERE id=OLD.student_id;
  ENDIF;
RET
URN
NULL;
END;
$$LANGUAGE plpgsql;

CREATE TRIGGER
update_student_credit_hours_trigger AFTER INSERT
OR DELETE ON enrollments
FOR each ROW EXECUTE FUNCTION update_student_credit_hours();

```

**63. Create a trigger to update the total credit hours earned by a student when they enroll or drop a course.**

Assuming you have a table instructors with columns id and total\_credit\_hours\_taught, create a trigger to update the total credit hours taught by an instructor when they are assigned or removed from a course: CREATE OR REPLACE FUNCTION Update\_instructor\_credit\_hours() RETURNS TRIGGER AS

```

$$BE
GIN
  IF(TG_OP='INSERT') THEN
    UPDATE instructors SET total_credit_hours_taught = total_credit_hours_taught
+NEW.credit_hours WHERE id = NEW.instructor_id;
  ELSIF (TG_OP = 'DELETE') THEN
    UPDATE instructors SET total_credit_hours_taught = total_credit_hours_taught
-OLD.credit_hours WHERE id=OLD.instructor_id;

```

```

    ENDIF
;RETURN

    NULL;

    END ;

$$LANGUAGEplpgsql;

```

```

CREATE                                TRIGGER
update_instructor_credit_hours_triggerAFTER  INSERT
OR DELETE ONcourses
FOREACHROWEXECUTEFUNCTIONupdate_instructor_credit_hours();

```

**64. Create a table for storing user login logs.**

```

CREATE                                TABLE
user_login_logs(idSERIAL
PRIMARYKEY ,
user_idINTNOTNULL,
login_timestampTIMESTAMPNOTNULL
);

```

**65. Write SQL queries to add data into the user login logs**

```

table.INSERT INTO user_login_logs (user_id,
login_timestamp)VALUES (1, '2023-03-3110:00:00');

```

```

INSERT INTO user_login_logs (user_id,
login_timestamp)VALUES (2, '2023-03-3110:05:00');

```

```

INSERT INTO user_login_logs (user_id,
login_timestamp)VALUES (1, '2023-03-3114:00:00');

```

**66. Write a query to retrieve a specific user's login logs By their user ID.**

```

SELECT *FROMuser_login_logs
WHERE

```

User\_id=1;--Replace 1 with the desired user ID



**67. Write a query to delete a specific user's login logs from the user\_login\_log table.**

```
DELETE FROM user_login_logs
WHERE user_id=1; -- Replace 1 with the desired user ID
```

**68. Create An Index on the student table to improve performance.**

```
CREATE INDEX idx_student_last_Name
ON student (last_name); -- Replace 'student' with your actual student table
and 'last_name' with the desired column
```

**69. Create an Index on the course table to improve query performance.**

```
CREATE INDEX idx_course_name
ON course (name); -- Replace 'course' with your actual course table and 'name' with the
desired column
```

**70. Analyze the performance of a specific query using the Oracle Explain Plan feature.**

To analyze the performance of a specific query, you can use the EXPLAIN PLAN statement in Oracle. Here is a case of how to use the EXPLAIN PLAN feature:

```
-- Replace the SELECT statement with your specific
query
EXPLAIN PLAN FOR
SELECT * FROM user_login_logs WHERE user_id=1;
```

```
-- To view the output of the EXPLAIN PLAN, you can query the
PLAN_TABLE:
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY());
```

**71. Use SQL subqueries to find students who are registered for the same course as a specific student.**

```
SELECT DISTINCT s2.student_id,
s2.student_name FROM course_registration AS
cr1
JOIN course_registration AS cr2 ON cr1.Course_id =
cr2.course_id JOIN student AS s2 ON cr2.student_id = s2.student_id
```

```
WHERE cr1.student_id = <specific_student_id> AND cr1.student_id != s2.student_id;
```

**72. Use SQL subqueries to find instructors who have taught the same course as a specific instructor.**

```
“SELECT DISTINCT i2.instructor_id,
i2.instructor_nameFROM courseAs c1
JOINcourseAS c2ONc1.Course_id=c2.course_id
JOINinstructorASi2ONc2.instructOr_Id = i2.instructor_id
WHERE Ec1.instructor_id=<specific_instructor_id>ANDc1.instructor_id!=i2.instructor_id;”
```

**73. Write a query to implement pagination for a large result set, such as a list of all students.**

```
SELECT * FROM
studentORDER BY
student_id
LIMIT <page_size> OFFSET <offset>;
```

Replace <page\_size> with the number of records per page and <offset> with the starting record number forthe page (e.g.,(page\_number - 1)\*page\_size).

**74. Write a query to implement full-text search for student names or course titles.**

```
--For
studentnamesSELECT*F
ROMstudent
WHEREto_tsvector('enGlish',student_name)@@to_tsquery('english', '<search_query>');
--For
coursetitlesSELECT*F
ROMcourse
WHERE to_tsvector('english', course_title) @@ to_tsquery('english',
'<search_query>');Replace<search_query>with the text you want to search for.
```

**75. Use SQL transactions to ensure data integrity when enrolling students in courses or updating theirGPA.**

```
-- Enrolling a student in a
courseBEGIN;
```

```

INSERT INTO Course_registration (student_id,
course_id)VALUES(<stuDent_id>,<course_id>);
UPDATEstudent
SET enrolled_courses = enrolled_courses +
1WHERE student_id =
<student_id>;COMMIT;

```

-- Updating a Student's

```
GPABEGIN;
```

```
UPDATEstudent
```

```
SET gpa=<new_gpa>
```

```
WHEREstudent_id=<student_id>;
```

-- Any other related updates should be placed here, e.g., updating class rank,  
etc.COMMIT;

## **76. Optimize a slow-running query using indexes, subqueries, or other query optimization techniques.**

Here's an example of optimizing a slow-running query that finds students enrolled in a precise course:

### **Original query:**

```

“SELECT *
FROM student
WHERE Student_id
IN
(SELECTstudent_id
FROMcourse_registration
WHERECourse_id=<specific_course_id>
);”

```

### **Optimized Query Using JOIN:**

```
“SELECT s.*
```

```
FROM
```

StudentASs

```
JOIN course_registration AS cr ON s.student_id =  
cr.student_id WHERE cr.course_id = <specific_course_id>;”
```

**Additionally, you could create an index on course\_registration(course\_id) and student(student\_id) to speed up the join operation:**

```
“CREATE INDEX idx_course_registration_course_id ON  
Course_registration(course_id);  
  
CREATE INDEX idx_student_id ON student(student_id);”
```

**77. Use SQL joins to recover a list of students and the course they are registered for.**

```
“SELECT s.student_id, s.student_name, c.course_id,  
c.course_title FROM student AS s  
JOIN course_registration AS cr ON s.student_id =  
cr.student_id JOIN course AS c ON cr.course_id = c.course_id;”
```

**78. Use SQL joins to recover a record of instructors and the courses they are teaching.**

```
“SELECT i.instructor_id, i.instructor_name, c.course_id,  
c.course_title FROM instructor AS i  
JOIN course AS c ON i.instructor_id = c.instructor_id;”
```

**79. Use SQL joins to recover a list of courses and their prerequisites.**

```
SELECT c1.course_id AS course_id, c1.course_title AS  
course_title, c2.course_id AS prerequisite_id, c2.course_title AS prerequisite_  
title  
FROM course AS c1  
JOIN course_prerequisite AS cp ON c1.course_id =  
cp.course_id JOIN course AS c2 ON cp.prerequisite_id =  
c2.course_id;
```

**80. Write down a SQL query to create a normalized schema for an online store, including tables for customers, products, orders, and order items. Ensure each table has appropriate primary and foreign keys.**

```
“CREATE TABLE Customers (  
customer_id INT PRIMARY KEY AUTO_INCREMENT,
```

```

first_name VARCHAR(250) NOT NULL,last_name VARCHAR(250) NOT
NULL,emailVARCHAR(250)UNIQUENOTNULL
);

```

```

CREATE TABLE Products (
Product_idINT PRIMARYKEY YAUTO_INCREMENT,
product_name VARCHAR(255) NOT NULL,

PriceDECIMAL(10,2)NOTNULL,
stock_levelINTNOTNULL
);

```

```

CREATE TABLE Orders (
order_idINT PRIMARYKEY AUTO_INCREMENT,
customer_id INT NOT NULL,order_date DATE NOT NULL,total_priceDECIMAL( 10
,2)NOTNULL,
FOREIGNKEY(customer_Id)REFERENCES Customers(customer_id)
);
CREATETABLEOrder_Items(

```

```

order_item_id INT PRIMARY KEY AUTO_INCREMENT,order_idINT NOTNULL,
product_id INT NOT NULL,quantityINTNOTNULL,
FOREIGN KEY (order_id) REFERENCES
Orders(order_Id),FOREiGNKEY(product_id)REFERENCESProducts(product_id),)”

```

**81. Write down a “SQL query “ to create an index On the product name in the "Products" table,assuming the product name column is called"product\_name".**

```

CREATE INDEXidx_product_nameONProducts(product_name);

```

**82. Write a “SQL query” to find the top 3 Best-selling products in the online store, based on the total quantity sold.**

```

“SELECT p.product_id, p.product_name, SUM(oi.quantity) as
total_soldFROMProductsp

```

```

JOIN          order_Items oi          ON          p.product_id          =
oi.product_id GROUP BY p.product_id, p.product_name
ORDER BY total_sold DESC LIMIT 3;”

```

**83. Write a “SQL query” to find the total revenue generated by the online store for a given date range, using the "Orders" and "Order\_Items" tables.**

```

SELECT SUM(o.total_price) as total_revenue FROM Orders o
WHERE o.order_date BETWEEN start_date AND end_date;

```

**84. Create a stored procedure to insert a new customer record into the "Customers" table, checking for duplicate email addresses and returning an appropriate error message if duplicate is found.**

```

DELIMITER //
CREATE PROCEDURE InsertCustomer(IN p_first_name VARCHAR(255),
IN
p_last_name VARCHAR(255), IN p_email VARCHAR(255), OUT p_status VARCHAR(255)
) BEGIN
DECLARE email_exists INT DEFAULT 0;

```

```
SELECT COUNT(*) INTO email_exists FROM Customers
WHERE email=p_email;
```

```
IF email_exists > 0 THEN
```

```
    SET p_status = 'Error: Email address already exists.'; ELSE
```

```
    INSERT INTO Customers (First_name, last_name,
    email) VALUES (p_First_name, p_last_name, p_email);
```

```
    SET p_status = 'Success: Customer added.'; ENDIF;
```

```
END//DELIMITER;
```

**85. Create a stored procedure to update the stock level meant for a product in the "Products" table, considering the stock level should not be less than zero.**

```
DELIMITER//
```

```
CREATE PROCEDURE Update STOCK Level (IN p_product_id INT,
```

```
    IN p_STOCK_level INT,
```

```
    OUT p_status VARCHAR(255)
```

```
) BEGIN
```

```
    IF p_stock_level < 0 THEN
```

```
        SET p_status = 'Error: Stock level cannot be negative.'; ELSE
```

```
        UPDATE products
```

```
        SET stock_level = p_stock_level WHERE product_id = p_product_id;
```

```
        SET p_status = 'Success: Stock level
```

```
        updated.'; ENDIF;
```

```
END//DELIMI
```

```
TER;
```

**86. Write a SQL query to implement a transaction that inserts a new order and its associated order items into the "Orders" and "Order\_Items" tables, ensuring data consistency.**

```
BEGIN TRANSACTION;
```

```
--Insert the new order into the Orderstable
```

```
INSERT INTO Orders (order_id, customer_id,
```

```
order_date)VALUES(NEW_ORDER_ID,CUSTOMER_ID,'Y
```

```
YYY-MM-DD');
```

```
-- Insert the associated order items into the Order_Items table
```

```
INSERT INTO Order_Items (order_id, product_id, quantity, price)
```

```
VALUES(NEW_ORDER_ID,PRODUCT_ID_1,QUANTITY_1,PRICE_1),
```

```
    (NEW_ORDER_ID, PRODUCT_ID_2, QUANTITY_2,
```

```
    PRICE_2),(NEW_ORDER_ID,PRODUCT_ID_3,QUANTI
```

```
TY_3,PRICE_3);
```

```
-- Check for errors and commit the transaction if no errors
```

```
occurredIF@@ERROR=0
```

```
    COMMIT
```

```
TRANSACTION;ELSE
```

```
    ROLL BACK TRANSACTION;
```

**87. Write a SQL query to Create a view that displays the whole revenue generated by each customer.**

```
“CREATE VIEW Customer_RevenueAS
```

```
SELECT  c.customer_id,  c.customer_name,  SUM(oi.price  *  oi.quantity)  AS
```

```
total_revenue FROM Customersc
```

```
JOIN          Order          so          ONc.Customer_id=o.customer_id
```



```
JOIN      Order_Items oI      ON      o.order_id      =
oi.order_id GROUP BY c.customer_id, c.customer_name;”
```

**88. Implement a trigger that automatically calculates and updates the total price of an order in the "Orders" table when a new record is inserted into the "Order\_Items" table.**

```
“CREATE TRIGGER Update_Order_Total AFTER INSERT ON Order_Items
FOR EACH ROW BEGIN
UPDATE Orders
SET total_price = total_price + (NEW.quantity * NEW.price) WHERE
order_id=NEW.order_id;
END;”
```

**89. Write a SQL query to create a schema for a many-to-many relationship among students and courses, using a junction table for enrollments.**

```
-- Create Students table “CREATE TABLE students (student_id INT
PRIMARY KEY,
student_name VARCHAR(255) NOT NULL
);
-- Create Courses table CREATE TABLE Courses
(course_id INT PRIMARY KEY,
course_name VARCHAR(255) NOT NULL
);

-- Create Enrollments junction table
```

```
CREATE TABLE Enrollments (student_idINT,
course_Id INT,enrollment_dateDATE,
PRIMARYKEY(student_id, course_id),
FOREIGN KEY (Student_id) REFERENCES Students
(student_id),FOREIGNKEY(course_id)REFERENCES Courses (course_id)
);”
```

**90. Write a SQL query to uncover the average grade of all students in a specific course, using the "Students", "Courses", and "Enrollments" tables.**

```
“SELECT AVG(Enrollments.grade) AS average_grade FROM Students
JOIN Enrollments ON Students.Student_id = Enrollments.student_id JOIN Courses ON
Enrollments.course_id = Courses.course_id WHERE Courses.course_name =
'SpecificCourseName';”
```

**91. Write down a SQL query to identify clients who have placed more than 5 orders in the online store,using the "Customers" and "Orders" tables.**

```
“SELECT Customers.customer_id, Customers.customer_name FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_idGROUP BY
Customers.customer_id, Customers.customer_name HAVING COUNT(Orders.order_id) >
5;
```

**92. Write a SQL query to Find the products that have not been ordered in the last 30 days,using the "Products", "Orders", and "OrderItems" tables.**

```
“SELECT Products.product_id, Products.product_name FROM Products
LEFT JOIN Order_Items ON Products.Product_id =Order_Items.product_id
LEFT JOIN Orders ON Order_Items.order_id=Orders.order_id
WHERE Orders.order_date< CURRENT_DATE - INTERVAL '30 days' OR
Orders.order_id ISNULL
GROUP BY Products.product_id,Products.product_name;”
```

**93. Implement a trigger that checks the stock point of a product before inserting a new record into the "Order\_Items" table. If the stock level is insufficient, cancel the insertion and return an error message.**

```
“DELIMITER//
CREATE TRIGGER
```

```

check_stock_level BEFORE INSERT ON Order_Items FOR EACH ROW
BEGIN
    DECLARE stock_level INT;SELECT stock INTO stock_level FROM Products
    WHERE Products.ProdUct_id=NEW.prodUct_id;

    IF stock_level<NEW.quantity THEN
        SIGNAL SQLSTATE 45000'
        SET MESSAGE_TEXT = 'Insufficient stock level. ';ENDIF;
    END;
//DELIMITER :”

```

**94. Create a stored procedure to calculate and update the total price of an order after adding,updating, or deleting an orderitem.**

```

CREATE PROCEDURE Update Total Price(IN order_id INT)BEGIN

```

```

    DECLARE tOtal_price DECIMAL(10,2);

    SELECT SuM ( Products.price * Order_Items.quantity) INTO
    toTal_priceFROM Order_Items
    JOIN Products ON Order_Items.product_id = Products.product_id
    WHERE Order_Items.order_id = order_id;

    UPDATE OrdErs
    SET tOtal_price = tOtal_price
    WHERE
    ORders.order_id = ORder_id;
END;

```

**95. Write a SQL query to Create an index on the "Orders" table to optimize the search for order placed within a specific date range.**

```

CREATE INDEX idx_orders_order_date ON Orders (order_date);

```

**96. Write a “SQL query” to find the total number of enrollments for each course, sorted by the number of enrollments in descending order.**

```
“SELECT Courses.course_Id, Courses.course_name,  
COUNT(Enrollments.student_id) as enrollment_count  
FROM Courses
```

```
JOIN Enrollment ON Courses.course_id = Enrollments.course_id  
GROUP BY Courses.course_id, Courses.course_name  
ORDER BY enrollment_count DESC;”
```

**97. Create a view to display the number of products sold in each product category, using the "Products" and "Order\_Items" tables.**

```
“CREATE VIEW products_sold_by_category AS  
SELECT Products.category, COUNT(Order_Items.product_id) as products_sold
```

```
FROM Products
JOIN Order_Items ON Products.product_id = Order_Items.product_id GROUP BY
products.category;”
```

**98. Implement a Foreign key constraint on the "Enrollments" table to ensure that a record can only be added if the corresponding student and course IDs exist in the "Students" and "Courses" tables.**

```
“ALTER TABLE Enrollments
ADD FOREIGN KEY (student_id) REFERENCES
students(student_id),ADDFOREIGNKEY(course_id)
REFERENCE Scourses(course_id);”
```

**99. Write a SQL query to identify the top three customers who have spent the most money on top of the online store, based on the total price of their orders.**

```
“SELECT Customers.customer_id, customers.customer_name, SUM(Orders.total_price)
astotal_spent
FROM Customers
JOIN Orders ON customers.customer_id = Orders.customer_idGROUP BY
customers.customer_id, Customers.customer_name ORDER BY Total_spent DESC
LIMIT 3;”
```

**100. Write down a SQL query to find the students who have taken at least 1 course from each department, using the "Students", "Courses", "Enrollments", and "Departments" tables. Assume there is a "department\_id" column in both the "Courses" and "Departments" tables.**

To find the students who have taken at least 1 course from each department, you can use the following SQL query:

```
WITH Department Courses AS(
    SELECT DISTINCT department_id, course_idFROM Courses
),
Student Courses AS(SELECT e.student_id, dc.department_id FROM Enrollments
```

```

JOIN CoursescONe.course_id =c.courSe_id
JOIN Department CoursesdcONc.course_id=dc.course_id),
Department CountsAS(
  SELECT COUNT(*) AS department_count FROM Departments
),
StudentDepartmentCounts AS(
  SELECT student_id, COUNT(DISTINCT department_id)
  AS student_department_count FROM Student Courses
  GROUPBY Student_id
)
SELECT s.student_id, s.name FROM Students s
JOIN Student Department Counts secONds.Student_id=sd.student_id
JOIN Department Counts cONsdc.Student_department_count=dc.department_count;

```