Srinivasa Ramanujan Institute of Technology
Rotarypuram Village, BK Samudram Mandal, Ananthapuramu -515701
Accredited by NBA & NAAC with "A" Grade, Affiliated to JNTUA, Ananthapuramu

II B.Tech I Sem

Regulation: R23

Database Management Systems Lab Manual

23CSE302

Department of CSE

SRIT

Syllabus:

SRINIVASA RAMANUJAN INSTITUTE OF TECHNOLOGY

Database Management Systems Lab

(Common to CSE and CSM)

II B. Tech – I Semester								SRIT R23
Course Code	**Category**	**Hours/Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
23CSE304	**PCC**	0	0	3	1.5	30	70	100

List of Experiments

Week	Title of the Experiment
1	a) Write SQL queries to CREATE TABLES for various databases using DDL commands (i.e. CREATE, ALTER, DROP, TRUNCATE). b) Write SQL queries to MANIPULATE TABLES for various databases using DML commands (i.e. INSERT, SELECT, UPDATE, DELETE,).
2	a) Queries (along with sub-Queries) using ANY, ALL, IN, EXISTS, NOTEXISTS, UNION, INTERSET, Constraints. Example - Select the roll number and name of the student who secured fourth rank in the class. b) Queries using Aggregate functions (COUNT, SUM, AVG, MAX and MIN), GROUP BY, HAVING and Creation and dropping of Views.
3	a) Write SQL queries to perform RELATIONAL SET OPERATIONS (i.e. UNION, UNION ALL, INTERSECT, MINUS, CROSS JOIN, NATURALJOIN). b) Queries using Conversion functions (to_char, to_number and to_date), string functions (Concatenation, lpad, rpad, ltrim, rtrim, lower, upper, initcap, length, substr and instr), date functions (Sysdate, next_day, add_months, last_day, months_between, least, greatest, trunc, round, to_char, to_date)
4	Write SQL queries to perform JOIN OPERATIONS (i.e. CONDITIONALJOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTERJOIN).
5	a) Create a simple PL/SQL program which includes declaration section, executable section and exception –Handling section (Ex. Student marks can be selected from the table and printed for those who secured first class and an exception can be raised if no records were found) b) Insert data into student table and use COMMIT, ROLLBACK and SAVEPOINT in PL/SQL block.
6	a) Develop a program that includes the features NESTED IF, CASE and CASE expression. The program can be extended using the NULLIF and COALESCE functions. b) Program development using WHILE LOOPS, numeric FOR LOOPS, nested loops using ERROR Handling, BUILT –IN Exceptions, USE defined Exceptions, RAISE-APPLICATION ERROR.
7	a) Programs development using creation of procedures, passing parameters IN and OUT of PROCEDURES. b) Program development using creation of stored functions, invoke functions in SQL Statements and write complex functions.
8	Develop programs using features parameters in a CURSOR, FOR UPDATE CURSOR, WHERE CURRENT of clause and CURSOR variables.
9	Develop Programs using BEFORE and AFTER Triggers, Row and Statement Triggers

	and INSTEAD OF Triggers.
10	Create a table and perform the search operation on table using indexing and non-indexing techniques.
11	A publishing company produces scientific books on various subjects. The books are written by authors who specialize in one particular subject. The company employs editors who, not necessarily being specialists in a particular area, each take sole responsibility for editing one or more publications. A publication covers essentially one of the specialist subjects and is normally written by a single author. When writing a particular book, each author works with on editor, but may submit another work for publication to be supervised by other editors. To improve their competitiveness, the company tries to employ a variety of authors, more than one author being a specialist in a particular subject for the above case study, do the following: 1. Analyze the data required. 2. Normalize the attributes. Create the logical data model using E-R diagrams
12	A new e-commerce startup needs a database to manage its products, customers, orders, and inventory. Hence develop the database with following: • Design the database schema using Entity-Relationship (ER) diagrams. • Create the database and tables using SQL. • Insert sample data into the tables. • Write SQL queries to retrieve specific information, such as all orders for a particular customer, the current inventory levels, etc. • Create views to simplify complex queries.

Reference Books/Lab Manuals:	
1	Oracle: The Complete Reference by Oracle Press.
2	Rick F Vander Lans, "Introduction to SQL", Fourth Edition, Pearson Education, 2007.
3	RamezElmasri, Shamkant, B. Navathe, "Database Systems", Pearson Education, 6th Edition, 2013.

Course Outcomes: At the end of the course, the student should have acquired the ability to	
CO1	Design the tables with constraints, and efficiently insert data using SQL commands. **Cognitive Level: L6**
CO2	Develop complex SQL queries with sub-queries, set operations, and aggregate functions. **Cognitive Level: L6**
CO3	Use SQL aggregate functions, GROUP BY, and HAVING clauses effectively. **Cognitive Level: L3**
CO4	Develop basic PL/SQL programs with exception handling, transaction control commands (COMMIT, ROLLBACK, SAVEPOINT) and handle specific query scenarios. **Cognitive Level: L6**
CO5	Implement conditional statements (NESTED IF, CASE) and loops (WHILE, FOR) in PL/SQL. **Cognitive Level: L3**
CO6	Apply stored procedures with parameters for enhanced data processing. **Cognitive Level: L3**

Experiment No:

1 A) Write SQL queries to CREATE TABLES for various databases using DDL commands (i.e. CREATE, ALTER, DROP, TRUNCATE).

CREATE TABLE

CREATE TABLE Employees (EmployeeID INT PRIMARY KEY,FirstName VARCHAR(50),LastName VARCHAR(50),Email VARCHAR(100),HireDate DATE,Salary DECIMAL(10, 2));

Output:

ALTER TABLE

Add a new column:

ALTER TABLE Employees ADD Department VARCHAR(50);

Output:

Modify an existing column:

ALTER TABLE Employees MODIFY Salary DECIMAL(12, 2);

Output:

Drop a column:

ALTER TABLE Employees DROP COLUMN Email;

Output:

DROP TABLE

DROP TABLE Employees;

Output:

TRUNCATE TABLE

TRUNCATE TABLE Employees;

Output:

1 B) Write SQL queries to MANIPULATE TABLES for various databases using DML commands (i.e. INSERT, SELECT, UPDATE, DELETE).

INSERT INTO

-- Insert one record

INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate, Salary, DepartmentID)VALUES (1, 'John', 'Doe', '2023-01-10', 55000, 2);

Output:

-- Insert multiple records

INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate, Salary, DepartmentID)VALUES

(2, 'Jane', 'Smith', '2023-02-05', 62000, 3),

(3, 'Alice', 'Johnson', '2023-03-12', 59000, 1);

Output:

SELECT

Select all columns:

SELECT * FROM Employees;

Output:

Select specific columns:

SELECT FirstName, LastName, Salary FROM Employees;

Output:

Select with a condition:

SELECT * FROM Employees WHERE DepartmentID = 2;

Output:

Select with sorting:

SELECT * FROM EmployeesORDER BY Salary DESC;

Output:

Select with aggregate function:

SELECT DepartmentID, AVG(Salary) AS AverageSalary FROM Employees GROUP BY DepartmentID;

Output:

UPDATE

Update one record:

UPDATE EmployeesSET Salary = 60000WHERE EmployeeID = 1;

Output:

Update multiple records:

UPDATE EmployeesSET Salary = Salary * 1.05WHERE DepartmentID = 2;

Output:

DELETE

Delete one record:

DELETE FROM Employees

WHERE EmployeeID = 3;

Output:

Delete multiple records based on a condition:

DELETE FROM EmployeesWHERE DepartmentID = 2;

Output:

2 A) Queries (along with sub-Queries) using ANY, ALL, IN, EXISTS, NOTEXISTS, UNION, INTERSET, Constraints. Example - Select the roll number and name of the student who secured fourth rank in the class.

Subquery Example using ANY

SELECT EmployeeID, FirstName, LastName FROM Employees WHERE Salary > ANY (SELECT Salary FROM Employees WHERE DepartmentID = 2);

Output:

Subquery Example using ALL

SELECT EmployeeID, FirstName, LastName FROM Employees WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DepartmentID = 3);

Output:

Subquery Example using IN

SELECT RollNumber, Name FROM Students WHERE DepartmentID IN (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'Computer Science');

Output:

Subquery Example using EXISTS

SELECT RollNumber, Name FROM Students s WHERE EXISTS (SELECT * FROM Enrollments e WHERE e.StudentID = s.RollNumber);

Output:

Subquery Example using NOT EXISTS

SELECT RollNumber, Name FROM Students s WHERE NOT EXISTS (SELECT * FROM Enrollments e WHERE e.StudentID = s.RollNumber);

Output:

UNION Example

SELECT EmployeeID, FirstName, LastName FROM Employees WHERE DepartmentID =2
UNION
SELECT EmployeeID, FirstName, LastName FROM Employees WHERE DepartmentID=3;

Output:

INTERSECT Example

SELECT EmployeeID, FirstName, LastName FROM Employees WHERE DepartmentID =1
INTERSECT
SELECT EmployeeID, FirstName, LastName FROM Employees WHERE DepartmentID=2;

Output:

Constraints Example

CREATE TABLE Students (RollNumber INT PRIMARY KEY, Name VARCHAR(50) NOT NULL, DepartmentID INT, Rank INT, CONSTRAINT chk_Rank CHECK (Rank > 0 AND Rank <= 100), FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID));

Output:

Select the roll number and name of the student who secured fourth rank in the class.

SELECT RollNumber, Name FROM Students WHERE Rank = 4;

Output:

Subquery with Aggregate and HAVING

SELECT DepartmentID FROM Employees GROUP BY DepartmentID HAVING AVG(Salary) > 50000;

Output:

2 B) Queries using Aggregate functions (COUNT, SUM, AVG, MAX and MIN), GROUP BY, HAVING and Creation and dropping of Views.

COUNT, SUM, AVG, MAX, MIN Examples

COUNT Example

SELECT COUNT(*) AS TotalEmployees FROM Employees;

Output:

SUM Example

SELECT SUM(Salary) AS TotalSalary FROM Employees;

Output:

AVG Example

SELECT AVG(Salary) AS AverageSalary FROM Employees;

Output:

MAX Example

SELECT MAX(Salary) AS MaxSalary FROM Employees;

Output:

MIN Example

SELECT MIN(Salary) AS MinSalary FROM Employees;

Output:

GROUP BY Example

Group by Department and Find the Total Salary in Each Department:

SELECT DepartmentID, SUM(Salary) AS TotalSalary FROM Employees GROUP BY DepartmentID;

Output:

Group by Department and Find the Average Salary in Each Department:

SELECT DepartmentID, AVG(Salary) AS AverageSalary FROM Employees GROUP BY DepartmentID;

Output:

HAVING Example

Find Departments where the Total Salary is Greater than 100,000:

SELECT DepartmentID, SUM(Salary) AS TotalSalary FROM Employees GROUP BY DepartmentID HAVING SUM(Salary) > 100000;

Output:

Find Departments where the Average Salary is More than 50,000:

SELECT DepartmentID, AVG(Salary) AS AverageSalary FROM Employees GROUP BY DepartmentID HAVING AVG(Salary) > 50000;

Output:

Creation of Views

Create a View to Show High-Salary Employees:

CREATE VIEW HighSalaryEmployees AS SELECT EmployeeID, FirstName, LastName, Salary FROM Employees WHERE Salary > 70000;

Output:

Now, you can query the view like a table:

SELECT * FROM HighSalaryEmployees;

Output:

Create a View to Show Employees with Their Department Name:

CREATE VIEW EmployeeDepartment AS SELECT e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName FROM Employees e JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

SELECT * FROM EmployeeDepartment;

Output:

Dropping a View

DROP VIEW HighSalaryEmployees;

Output:

Combining Aggregates with Grouping and Having

Find Departments with More Than 5 Employees and Total Salary Over 200,000:

SELECT DepartmentID, COUNT(*) AS EmployeeCount, SUM(Salary) AS TotalSalary FROM Employees GROUP BY DepartmentID HAVING COUNT(*) > 5 AND SUM(Salary) > 200000;

Output:

3 A) Write SQL queries to perform RELATIONAL SET OPERATIONS (i.e. UNION, UNION ALL, INTERSECT, MINUS, CROSS JOIN, NATURALJOIN).

UNION

SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 1
UNION
SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 2;

Output:

UNION ALL

SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 1
UNION ALL
SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 2;

Output:

INTERSECT

SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 1
INTERSECT
SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 2;

Output:

MINUS (EXCEPT)

SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 1
MINUS
SELECT FirstName, LastName FROM Employees WHERE DepartmentID = 2;

Output:

CROSS JOIN

SELECT e.FirstName, e.LastName, d.DepartmentName FROM Employees e CROSS JOIN Departments d;

Output:

NATURAL JOIN

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e NATURAL JOIN Departments d;

Output:

INNER JOIN (Additional)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

LEFT JOIN (LEFT OUTER JOIN)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

RIGHT JOIN (RIGHT OUTER JOIN)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e RIGHT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

3 B) Queries using Conversion functions (to_char, to_number and to_date), string functions (Concatenation, lpad, rpad, ltrim, rtrim, lower, upper, initcap, length, substr and instr), date functions (Sysdate, next_day, add_months, last_day, months_between, least, greatest, trunc, round, to_char, to_date)

Conversion Functions

TO_CHAR

Convert a date to a specific format:

SELECT TO_CHAR(SYSDATE, 'DD-MM-YYYY') AS CurrentDate FROM dual;

Output:

Convert a number to a string with formatting:

SELECT TO_CHAR(12345.678, '99999.99') AS FormattedNumber FROM dual;

Output:

TO_NUMBER

Convert a string to a number:

SELECT TO_NUMBER('12345') AS Number FROM dual;

Output:

TO_DATE

Convert a string to a date in a specific format:

SELECT TO_DATE('05-OCT-2024', 'DD-MON-YYYY') AS ConvertedDate FROM dual;

String Functions

Concatenation

SELECT 'Hello' || ' ' || 'World' AS Greeting FROM dual;

Output:

LPAD and RPAD

Pad the left side of a string:

SELECT LPAD('123', 5, '0') AS PaddedValue FROM dual;

Output:

Pad the right side of a string:

SELECT RPAD('123', 5, '*') AS PaddedValue FROM dual;

Output:

LTRIM and RTRIM

Trim characters from the left:

SELECT LTRIM('000123', '0') AS TrimmedValue FROM dual;

Output:

Trim characters from the right:

SELECT RTRIM('123***', '*') AS TrimmedValue FROM dual;

Output:

LOWER, UPPER, INITCAP

Convert a string to lowercase:

SELECT LOWER('HELLO WORLD') AS LowerCaseText FROM dual;

Output:

Convert a string to uppercase:

SELECT UPPER('hello world') AS UpperCaseText FROM dual;

Output:

Capitalize the first letter of each word:

SELECT INITCAP('hello world') AS CapitalizedText FROM dual;

Output:

LENGTH

SELECT LENGTH('Hello') AS StringLength FROM dual;

Output:

SUBSTR

SELECT SUBSTR('Hello World', 1, 5) AS Substring FROM dual;

Output:

INSTR

SELECT INSTR('Hello World', 'World') AS Position FROM dual;

Output:

Date Functions

SYSDATE

SELECT SYSDATE AS CurrentDate FROM dual;

Output:

NEXT_DAY

SELECT NEXT_DAY(SYSDATE, 'MONDAY') AS NextMonday FROM dual;

Output:

ADD_MONTHS

SELECT ADD_MONTHS(SYSDATE, 3) AS FutureDate FROM dual;

Output:

LAST_DAY

SELECT LAST_DAY(SYSDATE) AS LastDayOfMonth FROM dual;

Output:

MONTHS_BETWEEN

SELECT MONTHS_BETWEEN(TO_DATE('31-DEC-2024', 'DD-MON-YYYY'), SYSDATE) AS MonthsBetween FROM dual;

Output:

LEAST and GREATEST

Find the earliest (least) date:

SELECT LEAST(TO_DATE('01-JAN-2024', 'DD-MON-YYYY'), SYSDATE) AS EarliestDate FROM dual;

Output:

Find the latest (greatest) date:

SELECT GREATEST(TO_DATE('01-JAN-2024', 'DD-MON-YYYY'), SYSDATE) AS LatestDate FROM dual;

Output:

TRUNC and ROUND

Truncate the date to the first day of the month:

SELECT TRUNC(SYSDATE, 'MM') AS TruncatedDate FROM dual;

Output:

Round the date to the nearest day:

SELECT ROUND(SYSDATE, 'DD') AS RoundedDate FROM dual;

Output:

TO_CHAR and TO_DATE with Dates

Convert a date to a string in a specific format:

SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS') AS FormattedDate FROM dual;

Output:

Convert a string to a date:

SELECT TO_DATE('15-OCT-2024', 'DD-MON-YYYY') AS ConvertedDate FROM dual;

Output:

4) Write SQL queries to perform JOIN OPERATIONS (i.e. CONDITIONALJOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTERJOIN).

Conditional Join (INNER JOIN with a Condition)

SELECT e.EmployeeID, e.FirstName, e.Salary, d.DepartmentName FROM Employees e JOIN Departments d ON e.DepartmentID = d.DepartmentID AND e.Salary > 50000;

Output:

Equi Join (INNER JOIN)

SELECT e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName FROM Employees e INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

Left Outer Join (LEFT JOIN)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

Right Outer Join (RIGHT JOIN)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e
RIGHT JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

Full Outer Join (FULL JOIN)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e
FULL OUTER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

Output:

Cross Join (Cartesian Product)

SELECT e.EmployeeID, e.FirstName, d.DepartmentName FROM Employees e
CROSS JOIN Departments d;

Output:

5 A) Create a simple PL/SQL program which includes declaration section, executable section and exception –Handling section (Ex. Student marks can be selected from the table and printed for those who secured first class and an exception can be raised if no records were found)

Assumptions:

There is a table named Students with the following structure:

StudentID (NUMBER)

StudentName (VARCHAR2)

Marks (NUMBER)

Program

```
DECLARE
   -- Declaration Section
   v_student_id Students.StudentID%TYPE;
   v_student_name Students.StudentName%TYPE;
   v_marks Students.Marks%TYPE;
   v_count NUMBER := 0; -- Variable to track if records are found
   -- Exception for no records found
   e_no_records_found EXCEPTION;

BEGIN
   -- Executable Section
   DBMS_OUTPUT.PUT_LINE('Students who secured first class (marks >= 60):');
   -- Cursor to select students who secured first class
   FOR student_rec IN (SELECT StudentID, StudentName, Marks
                       FROM Students
                       WHERE Marks >= 60)
   LOOP
      v_count := v_count + 1; -- Increment counter when records are found
      -- Print student details
      DBMS_OUTPUT.PUT_LINE('Student ID: ' || student_rec.StudentID ||
                           ', Name: ' || student_rec.StudentName ||
                          ', Marks: ' || student_rec.Marks);
   END LOOP;
   -- If no records found, raise the exception
   IF v_count = 0 THEN
      RAISE e_no_records_found;
```

```
    END IF;
EXCEPTION
    -- Exception Handling Section
    WHEN e_no_records_found THEN
        DBMS_OUTPUT.PUT_LINE('No records found for students with first class marks.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
/
```

Output:

If there are students who scored first class:

Students who secured first class (marks >= 60):

Student ID: 1, Name: John Doe, Marks: 75

Student ID: 2, Name: Jane Smith, Marks: 85

If no students scored first class:

No records found for students with first class marks.

5 B) Insert data into student table and use COMMIT, ROLLBACK and SAVEPOINT in PL/SQL block.

Assumptions:

The Students table has the following structure:

StudentID (NUMBER)

StudentName (VARCHAR2)

Marks (NUMBER)

Program

```
BEGIN
    -- Inserting records into the Students table
    INSERT INTO Students (StudentID, StudentName, Marks)
    VALUES (1, 'John Doe', 75);
    -- Save the transaction at this point
    SAVEPOINT first_save;
    INSERT INTO Students (StudentID, StudentName, Marks)
    VALUES (2, 'Jane Smith', 85);
    -- Save another point after the second insertion
    SAVEPOINT second_save;
    INSERT INTO Students (StudentID, StudentName, Marks)
    VALUES (3, 'Alice Johnson', 50);
```

```
    -- Rollback to the second_save point, this will remove the record for Alice Johnson
    ROLLBACK TO second_save;
    -- Inserting another record after rollback
    INSERT INTO Students (StudentID, StudentName, Marks)
    VALUES (4, 'Bob Brown', 65);
    -- Commit the transaction to finalize changes
    COMMIT;
    -- Display a message indicating successful transaction
  DBMS_OUTPUT.PUT_LINE('Transaction committed successfully.');
  EXCEPTION
    -- Exception handling: rollback everything if any error occurs
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
        ROLLBACK; -- Rollback the entire transaction
END;
/
```

Output:

After running the block, you will see the message:

Transaction committed successfully.

Data After the Block Execution:

The Students table will have the following records after the COMMIT:

StudentID	StudentName	Marks
1	John Doe	75
2	Jane Smith	85
4	Bob Brown	65

6 A) Develop a program that includes the features NESTED IF, CASE and CASE expression. The program can be extended using the NULLIF and COALESCE functions.

The program evaluates the marks of a student and assigns a grade based on the following conditions:

Marks ≥ 90: Grade 'A'

Marks between 80 and 89: Grade 'B'

Marks between 70 and 79: Grade 'C'

Marks between 60 and 69: Grade 'D'

Marks < 60: Fail ('F')

Program

```
DECLARE
   v_student_name  VARCHAR2(50) := 'John Doe';
   v_marks         NUMBER := NULL;  -- Assuming NULL as initial marks value
   v_grade         CHAR(1);
   v_previous_marks NUMBER := 85;
   v_final_marks    NUMBER;
BEGIN
   -- Using COALaESCE to provide a default value for NULL marks
   v_marks := COALESCE(v_marks, 50);
-- If marks are NULL, assign a default value of 50
   -- NESTED IF to determine grade based on marks
   IF v_marks >= 90 THEN
      v_grade := 'A';
   ELSIF v_marks >= 80 THEN
      IF v_marks < 90 THEN
         v_grade := 'B';
      END IF;
   ELSIF v_marks >= 70 THEN
      v_grade := 'C';
   ELSIF v_marks >= 60 THEN
      v_grade := 'D';
   ELSE
      v_grade := 'F';  -- Fail
   END IF;
   -- CASE statement to handle grades assignment
   CASE
      WHEN v_grade = 'A' THEN
         DBMS_OUTPUT.PUT_LINE('Excellent! Grade A.');
      WHEN v_grade = 'B' THEN
         DBMS_OUTPUT.PUT_LINE('Very Good! Grade B.');
      WHEN v_grade = 'C' THEN
         DBMS_OUTPUT.PUT_LINE('Good! Grade C.');
      WHEN v_grade = 'D' THEN
         DBMS_OUTPUT.PUT_LINE('Fair! Grade D.');
      WHEN v_grade = 'F' THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Needs Improvement. Fail.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No grade available.');
  END CASE;
  -- CASE Expression to determine grade using CASE (alternative to NESTED IF)
  v_grade := CASE
            WHEN v_marks >= 90 THEN 'A'
            WHEN v_marks >= 80 THEN 'B'
            WHEN v_marks >= 70 THEN 'C'
            WHEN v_marks >= 60 THEN 'D'
            ELSE 'F'
          END;
  -- Output based on CASE expression
  DBMS_OUTPUT.PUT_LINE('Student ' || v_student_name || ' scored ' || v_marks || ' and
received Grade: ' || v_grade);
  -- Using NULLIF to set v_final_marks to NULL if previous marks and current marks are
the same
  v_final_marks := NULLIF(v_marks, v_previous_marks);
-- If v_marks = v_previous_marks, result will be NULL
  -- Output based on NULLIF result
  IF v_final_marks IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Marks have not changed.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Final marks: ' || v_final_marks);
  END IF;
EXCEPTION
  -- Exception handling
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
/
```

Output:

If the marks are 50 (due to COALESCE assigning the default value):

Fair! Grade D.

Student John Doe scored 50 and received Grade: D

Final marks: 50

If the v_marks and v_previous_marks are the same:

Fair! Grade D.

Student John Doe scored 50 and received Grade: D

Marks have not changed.

6 B) Program development using WHILE LOOPS, numeric FOR LOOPS, nested loops using ERROR Handling, BUILT –IN Exceptions, USE defined Exceptions, RAISE-APPLICATION ERROR.

Program

```
-- Create a user-defined exception
DECLARE
   -- User-defined exception
   invalid_input EXCEPTION;
   -- Procedure to process numbers
   PROCEDURE process_numbers(num IN NUMBER) IS
      v_square NUMBER;
   BEGIN
      -- Validate input
      IF num < 0 THEN
         RAISE invalid_input; -- Raise application error for negative input
      END IF;
      -- Numeric FOR Loop to calculate the square
      FOR i IN 1..num LOOP
         v_square := i * i;
         DBMS_OUTPUT.PUT_LINE('The square of ' || i || ' is ' || v_square);
      END LOOP;
      -- Nested Loop Example
      DBMS_OUTPUT.PUT_LINE('Creating a multiplication table:');
      FOR i IN 1..5 LOOP  -- Outer loop for numbers 1-5
         FOR j IN 1..5 LOOP  -- Inner loop for numbers 1-5
            DBMS_OUTPUT.PUT(i || ' x ' || j || ' = ' || (i * j) || CHAR(9));
         END LOOP;
         DBMS_OUTPUT.PUT_LINE('');  -- New line after each row of the table
      END LOOP;
   END process_numbers;
   -- Main block
   v_input NUMBER;
```

```
    v_exit NUMBER := 0; -- Variable to control loop

BEGIN
    WHILE v_exit != -1 LOOP
        BEGIN
            DBMS_OUTPUT.PUT('Enter a positive number (or -1 to quit): ');
            -- Reading input using the DBMS_OUTPUT is not possible.
            -- Using SQL*Plus or similar environments where we can read input.
            -- So, in an actual application, consider replacing this with a prompt or using an interface.
            v_input := &num;  -- Use substitution variable for input
                    -- Check for exit condition
            IF v_input = -1 THEN
                DBMS_OUTPUT.PUT_LINE('Exiting program.');
                v_exit := -1; -- Update exit variable
            ELSE
                -- Call the procedure to process the number
                process_numbers(v_input);
            END IF;
        EXCEPTION
            WHEN invalid_input THEN
                DBMS_OUTPUT.PUT_LINE('Error: Negative numbers are not allowed.');
            WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
        END;
    END LOOP;

END;
/
```

To run the program:

Use an Oracle database environment that supports PL/SQL (like SQL*Plus, Oracle SQL Developer, or another compatible IDE).

You may need to adjust the input mechanism based on your environment since DBMS_OUTPUT.PUT does not allow for user input. In this example, a substitution variable &num is used to illustrate how input could be handled in a typical interactive environment.

Output:

7 A) Programs development using creation of procedures, passing parameters IN and OUT of PROCEDURES.

Program

```
-- Create a package to encapsulate the procedures
CREATE OR REPLACE PACKAGE number_utils AS
   PROCEDURE calculate_square(p_number IN NUMBER, p_square OUT NUMBER);
   PROCEDURE calculate_sum(p_number1 IN NUMBER, p_number2 IN NUMBER, p_sum OUT NUMBER);
END number_utils;
/
-- Package Body
CREATE OR REPLACE PACKAGE BODY number_utils AS
   -- Procedure to calculate the square of a number
   PROCEDURE calculate_square(p_number IN NUMBER, p_square OUT NUMBER) IS
   BEGIN
      p_square := p_number * p_number;
   END calculate_square;
   -- Procedure to calculate the sum of two numbers
   PROCEDURE calculate_sum(p_number1 IN NUMBER, p_number2 IN NUMBER, p_sum OUT NUMBER) IS
   BEGIN
      p_sum := p_number1 + p_number2;
   END calculate_sum;
END number_utils;
/
-- Main Block to execute the procedures
DECLARE
   v_square NUMBER;
   v_sum NUMBER;
   v_num1 NUMBER := 4;
   v_num2 NUMBER := 5;
BEGIN
   -- Calling the procedure to calculate square
   number_utils.calculate_square(v_num1, v_square);
   DBMS_OUTPUT.PUT_LINE('The square of ' || v_num1 || ' is ' || v_square);
```

```
    -- Calling the procedure to calculate sum
    number_utils.calculate_sum(v_num1, v_num2, v_sum);
    DBMS_OUTPUT.PUT_LINE('The sum of ' || v_num1 || ' and ' || v_num2 || ' is ' || v_sum);
END;
/
```

Execution

To run this program:

Use an Oracle database environment that supports PL/SQL (like SQL*Plus, Oracle SQL Developer, or another compatible IDE).

Make sure DBMS_OUTPUT is enabled to view the output in your environment.

Output:

7 B) Program development using creation of stored functions, invoke functions in SQL Statements and write complex functions.

Program

Creating Stored Functions

```
-- Create a function to calculate factorial
CREATE OR REPLACE FUNCTION calculate_factorial (p_number IN NUMBER) RETURN NUMBER IS
    v_result NUMBER := 1;
BEGIN
    IF p_number < 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Factorial is not defined for negative numbers.');
    ELSIF p_number = 0 THEN
        RETURN 1; -- 0! is 1
    ELSE
        FOR i IN 1..p_number LOOP
            v_result := v_result * i;
        END LOOP;
        RETURN v_result;
    END IF;
END calculate_factorial;
/

-- Create a function to concatenate first and last names
```

```
CREATE OR REPLACE FUNCTION get_full_name (p_first_name IN VARCHAR2, p_last_name IN VARCHAR2) RETURN VARCHAR2 IS
BEGIN
   RETURN p_first_name || ' ' || p_last_name;
END get_full_name;
/
```

Output:

Invoking Functions in SQL Statements

```
-- Main Block to demonstrate function invocation
DECLARE
   v_factorial NUMBER;
   v_full_name VARCHAR2(100);
BEGIN
   -- Calling the factorial function
   v_factorial := calculate_factorial(5);
   DBMS_OUTPUT.PUT_LINE('The factorial of 5 is ' || v_factorial);

   -- Calling the full name function
   v_full_name := get_full_name('John', 'Doe');
   DBMS_OUTPUT.PUT_LINE('The full name is ' || v_full_name);
END;
/
```

Output:

Using Functions in SQL Queries

```
-- SQL to use the calculate_factorial function
SELECT
   level AS number,
   calculate_factorial(level) AS factorial_value
FROM
   dual
CONNECT BY
   level <= 5;

-- SQL to use the get_full_name function
SELECT
   get_full_name('Alice', 'Smith') AS full_name
```

```
FROM
   dual;
```

Output:

8) Develop programs using features parameters in a CURSOR, FOR UPDATE CURSOR, WHERE CURRENT of clause and CURSOR variables.

Program

Creating a Table for Demonstration

```
-- Create a sample table
CREATE TABLE employees (
   employee_id NUMBER PRIMARY KEY,
   first_name VARCHAR2(50),
   last_name VARCHAR2(50),
   salary NUMBER
);
-- Insert some sample data
INSERT INTO employees VALUES (1, 'John', 'Doe', 50000);
INSERT INTO employees VALUES (2, 'Jane', 'Smith', 60000);
INSERT INTO employees VALUES (3, 'Jim', 'Brown', 55000);
INSERT INTO employees VALUES (4, 'Alice', 'Johnson', 70000);
COMMIT;
```

Using Parameters in Cursors, FOR UPDATE Cursors, and WHERE CURRENT OF Clause

```
DECLARE
   -- Cursor with parameters
   CURSOR emp_cursor (min_salary NUMBER) IS
      SELECT employee_id, first_name, last_name, salary
      FROM employees
      WHERE salary > min_salary
      FOR UPDATE; -- FOR UPDATE cursor

   v_employee_id employees.employee_id%TYPE;
   v_first_name employees.first_name%TYPE;
   v_last_name employees.last_name%TYPE;
   v_salary employees.salary%TYPE;

BEGIN
   -- Open the cursor for a minimum salary of 55000
```

```
    OPEN emp_cursor(55000);

    LOOP
        -- Fetch the current row into variables
        FETCH emp_cursor INTO v_employee_id, v_first_name, v_last_name, v_salary;

        -- Exit loop if no more rows
        EXIT WHEN emp_cursor%NOTFOUND;

        -- Print employee details
        DBMS_OUTPUT.PUT_LINE('Updating employee: ' || v_first_name || ' ' || v_last_name
|| ' with salary: ' || v_salary);

        -- Increase the salary by 10%
        v_salary := v_salary * 1.10;

        -- Update the current row using WHERE CURRENT OF
        UPDATE employees
        SET salary = v_salary
        WHERE CURRENT OF emp_cursor;
    END LOOP;

    -- Close the cursor
    CLOSE emp_cursor;

    -- Display updated employee details
    DBMS_OUTPUT.PUT_LINE('Employees updated successfully.');
END;
/
```

Output:

9) Develop Programs using BEFORE and AFTER Triggers, Row and Statement Triggers and INSTEAD OF Triggers.

Program

Creating a Sample Table

```
-- Create a sample table
CREATE TABLE employees (
```

```
    employee_id NUMBER PRIMARY KEY,
    first_name VARCHAR2(50),
    last_name VARCHAR2(50),
    salary NUMBER
);

-- Insert some sample data
INSERT INTO employees VALUES (1, 'John', 'Doe', 50000);
INSERT INTO employees VALUES (2, 'Jane', 'Smith', 60000);
COMMIT;
```

BEFORE and AFTER Triggers

BEFORE Trigger

```
-- Create a BEFORE trigger to log the employee details before insertion
CREATE OR REPLACE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inserting Employee: ' || :NEW.first_name || ' ' ||
:NEW.last_name);
END;
/
```

AFTER Trigger

```
-- Create an AFTER trigger to log the employee details after insertion
CREATE OR REPLACE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee Inserted: ' || :NEW.first_name || ' ' ||
:NEW.last_name);
END;
/
```

Row and Statement Triggers

```
-- Create an AFTER statement trigger
CREATE OR REPLACE TRIGGER after_employee_insert_statement
AFTER INSERT ON employees
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('A new employee record has been inserted.');
END;
/
```

INSTEAD OF Trigger

Create a View

```
CREATE OR REPLACE VIEW employee_view AS
SELECT employee_id, first_name, last_name FROM employees;
```

INSTEAD OF Trigger

```
-- Create an INSTEAD OF trigger on the view
CREATE OR REPLACE TRIGGER instead_of_employee_insert
INSTEAD OF INSERT ON employee_view
FOR EACH ROW
BEGIN
    INSERT INTO employees (employee_id, first_name, last_name, salary)
    VALUES (:NEW.employee_id, :NEW.first_name, :NEW.last_name, 50000); -- Default salary
END;
/
```

Testing the Triggers

```
-- Insert a new employee
INSERT INTO employees (employee_id, first_name, last_name, salary) VALUES (3, 'Alice', 'Johnson', 70000);
COMMIT;
-- Insert into the view
INSERT INTO employee_view (employee_id, first_name, last_name) VALUES (4, 'Bob', 'Martin');
COMMIT;
```

Output:

10) Create a table and perform the search operation on table using indexing and non-indexing techniques.

Program

Creating a Table

```
-- Create a sample table
CREATE TABLE employees (
    employee_id NUMBER PRIMARY KEY,
    first_name VARCHAR2(50),
```

```
    last_name VARCHAR2(50),
    department_id NUMBER,
    salary NUMBER
);

-- Insert some sample data
INSERT INTO employees VALUES (1, 'John', 'Doe', 101, 50000);
INSERT INTO employees VALUES (2, 'Jane', 'Smith', 102, 60000);
INSERT INTO employees VALUES (3, 'Jim', 'Brown', 101, 55000);
INSERT INTO employees VALUES (4, 'Alice', 'Johnson', 103, 70000);
INSERT INTO employees VALUES (5, 'Bob', 'Martin', 102, 62000);
COMMIT;
```

Creating an Index

```
-- Create an index on the last_name column
CREATE INDEX idx_last_name ON employees(last_name);
```

Search Operations

```
-- Non-indexed search using the last_name column
SELECT * FROM employees WHERE last_name = 'Doe';
-- Indexed search using the last_name column
SELECT * FROM employees WHERE last_name = 'Smith';
```

Performance Comparison

Check Execution Plan: You can use the EXPLAIN PLAN statement to see how the database will execute the queries.

```
EXPLAIN PLAN FOR
SELECT * FROM employees WHERE last_name = 'Doe';
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

EXPLAIN PLAN FOR
SELECT * FROM employees WHERE last_name = 'Smith';
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

Output:

11) A publishing company produces scientific books on various subjects. The books are written by authors who specialize in one particular subject. The company employs editors who, not necessarily being specialists in a particular area, each take sole responsibility for editing one or more publications. A publication covers essentially one of the specialist subjects and is normally written by a single author. When writing a particular book, each

author works with on editor, but may submit another work for publication to be supervised by other editors. To improve their competitiveness, the company tries to employ a variety of authors, more than one author being a specialist in a particular subject for the above case study, do thefollowing:

1. Analyze the datarequired.
2. Normalize theattributes.

Create the logical data model using E-R diagrams

Solution

1. Analyze the Data Required

The system involves several key entities and relationships based on the case study:

- **Author**: An individual who specializes in a particular subject and writes books.
- **Book (Publication)**: A scientific book that covers one subject and is written by one author.
- **Editor**: An employee responsible for editing books. Each editor may edit multiple books but does not necessarily specialize in a subject.
- **Subject**: Represents a scientific area of specialization. Multiple authors may specialize in the same subject.

Relationships:

- Each book is written by a single author.
- Each book is edited by one editor.
- An author can write multiple books, which may be edited by different editors.
- An editor can edit multiple books.
- Authors specialize in one subject, but multiple authors can specialize in the same subject.

2. Normalize the Attributes

To normalize the data, we'll break down the information into several tables/entities while ensuring no redundancy, proper dependencies, and normalization to at least the third normal form (3NF).

Entities and Attributes:

- **Author**:
  - `AuthorID` (Primary Key)
  - `Name`
  - `SpecializationID` (Foreign Key to Subject)
- **Book**:
  - `BookID` (Primary Key)
  - `Title`
  - `SubjectID` (Foreign Key to Subject)
  - `AuthorID` (Foreign Key to Author)
  - `EditorID` (Foreign Key to Editor)
- **Editor**:
  - `EditorID` (Primary Key)
  - `Name`
- **Subject**:
  - `SubjectID` (Primary Key)

- SubjectName

3. Create the Logical Data Model Using E-R Diagrams

Here's how the E-R diagram can be structured:

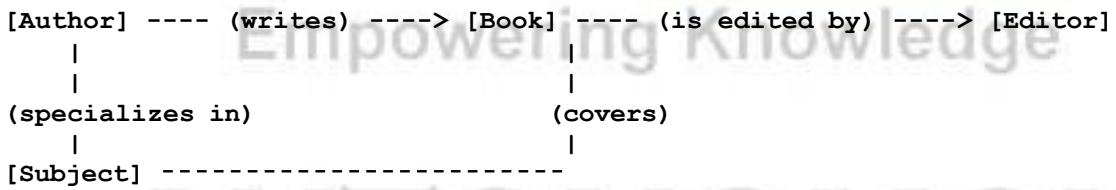
Entities:

1. **Author**:
   - Attributes: `AuthorID`, `Name`, `SpecializationID`
   - Relationship: Writes one or more books.
2. **Book**:
   - Attributes: `BookID`, `Title`, `SubjectID`, `AuthorID`, `EditorID`
   - Relationships:
     - Each book is written by one author.
     - Each book is edited by one editor.
     - Each book covers one subject.
3. **Editor**:
   - Attributes: `EditorID`, `Name`
   - Relationship: Edits one or more books.
4. **Subject**:
   - Attributes: `SubjectID`, `SubjectName`
   - Relationship: Multiple authors can specialize in one subject.

E-R Diagram:

1. **Author** (1) → (writes) → **Book** (M)
   - One author can write multiple books, but each book is written by a single author.
2. **Book** (M) → (is edited by) → **Editor** (1)
   - One book is edited by a single editor, but each editor can edit multiple books.
3. **Book** (M) → (covers) → **Subject** (1)
   - Each book covers one subject, but each subject can have multiple books.
4. **Author** (M) → (specializes in) → **Subject** (1)
   - Multiple authors can specialize in the same subject.

E-R Diagram Illustration:



Explanation of Relationships:

- **Author-Book**: A one-to-many relationship where an author can write many books, but each book is written by a single author.
- **Book-Editor**: A many-to-one relationship where a book is edited by one editor, but each editor can edit multiple books.
- **Book-Subject**: A many-to-one relationship where each book covers one subject, but multiple books can cover the same subject.
- **Author-Subject**: A many-to-one relationship where multiple authors can specialize in the same subject.

This model captures all the requirements of the case study, ensures normalized attributes, and organizes the relationships between the entities effectively.

12) A new e-commerce startup needs a database to manage its products, customers, orders, and inventory. Hence develop the database with following:

- Design the database schema using Entity-Relationship (ER) diagrams.
- Create the database and tables using SQL.
- Insert sample data into the tables.
- Write SQL queries to retrieve specific information, such as all orders for a particular customer, the current inventory levels, etc.
- Create views to simplify complex queries.

Program

To address the e-commerce startup's requirements for managing products, customers, orders, and inventory, I will walk through each step, from designing the database schema to writing SQL queries and creating views. Let's break it down:

1. Database Schema Design Using ER Diagrams

Key entities include:

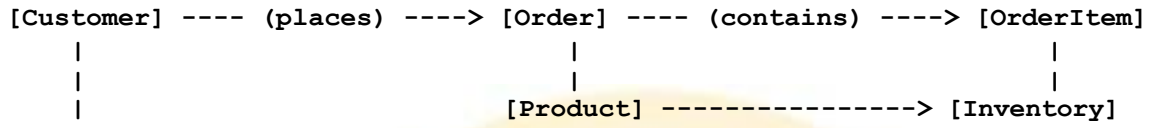
- **Customer**: Represents individuals purchasing from the site.
- **Product**: Represents items sold by the startup.
- **Order**: Represents customer orders.
- **OrderItem**: Represents individual items within an order.
- **Inventory**: Tracks stock levels for each product.

Entities and Attributes:

1. **Customer**:
   - CustomerID (Primary Key)
   - Name
   - Email
   - Phone
   - Address
2. **Product**:
   - ProductID (Primary Key)
   - Name
   - Description
   - Price
3. **Order**:
   - OrderID (Primary Key)
   - CustomerID (Foreign Key to Customer)
   - OrderDate
   - TotalAmount
4. **OrderItem**:
   - OrderItemID (Primary Key)
   - OrderID (Foreign Key to Order)
   - ProductID (Foreign Key to Product)
   - Quantity
   - Price (Price of the product at the time of order)
5. **Inventory**:

- InventoryID (Primary Key)
- ProductID (Foreign Key to Product)
- StockQuantity

ER Diagram Illustration:



Explanation of Relationships:

- **Customer-Order**: One customer can place many orders, but each order is placed by one customer.
- **Order-OrderItem**: Each order can contain multiple products (through OrderItem), and each product can appear in multiple orders.
- **Product-Inventory**: Each product has a stock quantity, which is tracked in the Inventory entity.

2. Create the Database and Tables Using SQL

Here is the SQL code to create the necessary tables:

```
-- Create Customer Table
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Phone VARCHAR(20),
    Address VARCHAR(255)
);

-- Create Product Table
CREATE TABLE Product (
    ProductID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100),
    Description TEXT,
    Price DECIMAL(10, 2)
);

-- Create Order Table
CREATE TABLE `Order` (
    OrderID INT PRIMARY KEY AUTO_INCREMENT,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);

-- Create OrderItem Table
CREATE TABLE OrderItem (
    OrderItemID INT PRIMARY KEY AUTO_INCREMENT,
    OrderID INT,
    ProductID INT,
    Quantity INT,
```

```
    Price DECIMAL(10, 2),
    FOREIGN KEY (OrderID) REFERENCES `Order`(OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);

-- Create Inventory Table
CREATE TABLE Inventory (
    InventoryID INT PRIMARY KEY AUTO_INCREMENT,
    ProductID INT,
    StockQuantity INT,
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
```

Output:

3. Insert Sample Data into the Tables

Now, let's populate the tables with some sample data:

```
-- Insert Customers
INSERT INTO Customer (Name, Email, Phone, Address)
VALUES
('John Doe', 'john@example.com', '555-1234', '123 Main St'),
('Jane Smith', 'jane@example.com', '555-5678', '456 Oak Ave');

-- Insert Products
INSERT INTO Product (Name, Description, Price)
VALUES
('Laptop', 'A high-performance laptop', 999.99),
('Smartphone', 'A new generation smartphone', 699.99),
('Headphones', 'Noise-cancelling headphones', 199.99);

-- Insert Orders
INSERT INTO `Order` (CustomerID, OrderDate, TotalAmount)
VALUES
(1, '2024-10-01', 1699.98),
(2, '2024-10-02', 699.99);

-- Insert Order Items
INSERT INTO OrderItem (OrderID, ProductID, Quantity, Price)
VALUES
(1, 1, 1, 999.99),
(1, 2, 1, 699.99),
(2, 2, 1, 699.99);

-- Insert Inventory Data
INSERT INTO Inventory (ProductID, StockQuantity)
VALUES
(1, 50),  -- Laptop
(2, 100), -- Smartphone
(3, 200); -- Headphones
```

Output:

4. Write SQL Queries to Retrieve Specific Information

a. Retrieve all orders for a particular customer:

```
SELECT O.OrderID, O.OrderDate, O.TotalAmount
FROM `Order` O
JOIN Customer C ON O.CustomerID = C.CustomerID
WHERE C.Name = 'John Doe';
```

Output:

b. Retrieve the current inventory levels:

```
SELECT P.Name, I.StockQuantity
FROM Inventory I
JOIN Product P ON I.ProductID = P.ProductID;
```

Output:

c. Retrieve the details of all products in an order:

```
SELECT O.OrderID, P.Name, OI.Quantity, OI.Price
FROM OrderItem OI
JOIN Product P ON OI.ProductID = P.ProductID
JOIN `Order` O ON OI.OrderID = O.OrderID
WHERE O.OrderID = 1;
```

Output:

5. Create Views to Simplify Complex Queries

a. Create a view to show order details with customer information:

```
CREATE VIEW CustomerOrderDetails AS
SELECT C.Name AS CustomerName, O.OrderID, O.OrderDate, O.TotalAmount
FROM Customer C
JOIN `Order` O ON C.CustomerID = O.CustomerID;
```

Output:

You can now retrieve customer order details using this simplified view:

```
SELECT * FROM CustomerOrderDetails WHERE CustomerName = 'John Doe';
```

Output:

b. Create a view for current stock levels:

```
CREATE VIEW ProductInventory AS
SELECT P.Name AS ProductName, I.StockQuantity
FROM Product P
JOIN Inventory I ON P.ProductID = I.ProductID;
```

Output:

You can now easily check inventory levels with:

```
SELECT * FROM ProductInventory;
```

Output: