Introduction DBMS

Introduction to SQL

Theory Questions:

1. What is SQL, and why is it essential in database management?

→

SQL (Structured Query Language) is a standard language used to manage and manipulate relational databases. It enables users to create, retrieve, update, and delete data efficiently. SQL is essential in database management because it provides a structured way to interact with databases, ensuring data integrity, security, and efficient data handling.

2. Explain the difference between DBMS and RDBMS.

→

DBMS:

- Database Management System.
- Stores data in a hierarchical or network model (e.g., file systems)
- Does not enforce relationships between data
- Less normalized, which can lead to redundancy
- EX - Microsoft Access, XML Databases

RDBMS:

- Stores data in tables with rows and columns
- Maintains relationships using primary keys and foreign keys
- Supports normalization to eliminate redundancy
- Fully supports ACID (Atomicity, Consistency, Isolation, Durability) properties
- MySQL, PostgreSQL, Oracle, SQL Server

3. Describe the role of SQL in managing relational databases.

→

A) **DDL - Data Definition Language**:
- Create, alter, and delete tables (CREATE, ALTER, DROP).

B) DML - Data Manipulation Language:
- Insert, update, delete, and retrieve records (INSERT, UPDATE, DELETE seLECT).

C) DCL - Data Control Language:
- Manage permissions and user access (GRANT, REVOKE).

D) TCL - Transaction Control Language: Manage transactions (COMMIT, ROLLBACK, SAVEPOINT).

4. What are the key features of SQL?

→

A) Data Querying: Retrieve data using SELECT statements.
B) Data Manipulation: Modify existing data using INSERT, UPDATE, and DELETE.
C) Data Definition: Define database schema using CREATE, ALTER, and DROP.
D) Data Integrity: Maintain consistency with constraints like PRIMARY KEY, FOREIGN KEY, UNIQUE, and CHECK.
E) Transaction Control: Support for COMMIT, ROLLBACK, and SAVEPOINT for data consistency.
F) Security: Control user access with GRANT and REVOKE commands.
G) Scalability: Efficiently handles large datasets and supports indexing for fast retrieval.

LAB EXERCISES:

Lab 1: Create a new database named school_db and a table called students with the following columns: student_id, student_name, age, class, and address.

→ -- Create the database
```
CREATE DATABASE school_db;

-- Create the students table
CREATE TABLE students (
    student_id INT PRIMARY KEY AUTO_INCREMENT,
    student_name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    class VARCHAR(20) NOT NULL,
    address TEXT
);
```

Lab 2: Insert five records into the students table and retrieve all records using the SELECT statement.

→ -- Insert five records into the students table
```
INSERT INTO students (student_name, age, class, address)
VALUES
(‘sandy', 14, '8th Grade', ‘neyork'),
('Jmina', 13, '7th Grade', ‘india'),
(‘dora', 15, '9th Grade', ‘ japan'),
(‘ella', 12, '6th Grade', ‘korea'),
(‘chutki', 14, '8th Grade', ‘india');

-- Retrieve all records
SELECT * FROM students;
```

2. SQL Syntax

Theory Questions:

1. What are the basic components of SQL syntax?

→

A) **Keywords**: Reserved words used to perform operations (e.g., SELECT, INSERT, UPDATE, DELETE, CREATE).
B) **Identifiers**: Names given to databases, tables, and columns (e.g., students, student_id).
C) **Data Types**: Define the type of data a column can hold (e.g., VARCHAR, INT, DATE).
D) **Operators**: Used to perform operations on data (=, >, <, LIKE, AND, OR).
E) **Clauses**: Modify the behavior of SQL statements (WHERE, ORDER BY, GROUP BY, HAVING)
F) .**Functions**: Perform calculations on data (COUNT(), SUM(), AVG()).
G) **Comments**: Explain SQL code without affecting execution (-- Single-line comment, /* Multi-line comment */).

2. Write the general structure of an SQL SELECT statement.

→

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column_name
HAVING condition
ORDER BY column_name ASC|DESC;
```

3. Explain the role of clauses in SQL statements.

→

A) **WHERE**Filters records based on conditions (WHERE age > 13).
B) **ORDER BY**Sorts results (ORDER BY student_name ASC).
C) **GROUP BY**Groups rows with the same values (GROUP BY class).
D) **HAVING**Filters grouped data (HAVING COUNT(*) > 2).
E) **LIMIT**Restricts the number of records returned (LIMIT 5).

LAB EXERCISES:

Lab 1: Write SQL queries to retrieve specific columns (student_name and age) from the students table.

→

```
SELECT student_name, age
FROM students;
```

output :-

student_name	age
John Doe	29
Jane Smith	12
Sam Wilson	16
Emily Johnson	08

Lab 2: Write SQL queries to retrieve all students whose age is greater than 10.

→

```
SELECT *
FROM students
WHERE age > 10;
```

Output :-

Student id	student name	age	class	address
1	sandy		8	newyork
2	mina		7	india
3	dora		12	japan
4	ella		10	lorea
5	chutki		11	india

3. SQL Constraints

Theory Questions:

1) What are constraints in SQL? List and explain the different types of constraints.

→

Constraints in SQL are rules applied to table columns to ensure the accuracy, consistency, and integrity of the data. They help enforce business rules at the database level, preventing invalid data entry.

→ Types :- A) Pri,ary key B) FoREIGN KEY C)NOTNULL D) CHECK E) UNIQUE E)DEFAULT

2. How do PRIMARY KEY and FOREIGN KEY constraints differ?

→

A) PRIMARY KEY
- Uniquely identifies each row in a table.
- Must be unique and not NULL.
- Cannot contain NULL values.
- Only one primary key per table (can be composite).
- Exists within its own table.

B) FOREIGN KEY
- Establishes a link between two tables.
- Can have duplicate values in the referencing table.
- Can contain NULL values if not enforced.
- Multiple foreign keys are allowed.
- References a primary key from another table.

3. What is the role of NOT NULL and UNIQUE constraints?

→

NOT NULL
- Ensures a column cannot store NULL values.

UNIQUE
- Ensures all values in a column are distinct (but allows NULL values).

LAB EXERCISES:

Lab 1: Create a table teachers with the following columns:
teacher_id (Primary Key), teacher_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).

→

```
CREATE TABLE teachers (
teacher_id INT PRIMARY KEY AUTO_INCREMENT,
teacher_name VARCHAR(100) NOT NULL,
subject VARCHAR(50) NOT NULL,
email VARCHAR(100) UNIQUE
);
```

Lab 2: Implement a FOREIGN KEY constraint to relate the teacher_id from the teachers table with the students table.

→

```
ALTER TABLE students
ADD COLUMN teacher_id INT,
ADD CONSTRAINT fk_teacher
FOREIGN KEY (teacher_id)
REFERENCES teachers(teacher_id)
ON DELETE CASCADE;
```

4. Main SQL Commands and Sub-commands (DDL)

Theory Questions:

1. Define the SQL Data Definition Language (DDL).

→ **SQL Data Definition Language (DDL)** consists of SQL commands used to define, modify, and delete database structures such as tables, schemas, and indexes.

→ DDL statements do not manipulate data but define its structure.

→ **Common DDL Commands:**

A) **CREATE**
- Used to create databases, tables, views, and indexes.

B) ALTER
- Modifies an existing table (e.g., add, delete, modify columns).

C) DROP
- Deletes an entire database, table, or index.

D) TRUNCATE
- Removes all records from a table but retains its structure.

2. Explain the CREATE command and its syntax

→ The **CREATE** command is used to define a new database, table, or other database objects.

→ Syntax :

```
CREATE TABLE table_name (
column1 datatype constraints,
column2 datatype constraints,
...
);
```

3. What is the purpose of specifying data types and constraints during table creation?

→ Defining **data types and constraints** is crucial in maintaining **data integrity, accuracy, and efficiency** in a database.

◉ → **Purpose of Data Types:**

◉ **Ensure Correct Data Storage** → Prevents incorrect data entry (e.g., storing text in a numeric column).

◉ **Optimize Storage Space** → Uses appropriate data types (INT, VARCHAR, etc.).

◉ **Improve Performance** → Helps in efficient indexing and query execution.

-- > Constainer	PURPOSE
PRIMARY KEY	Ensures each record has a unique identifier.
FOREIGN KEY	Establishes relationships between tables.
NOT NULL	Prevents NULL values in essential fields.
UNIQUE	Ensures no duplicate values in a column.
CHECK	Restricts values based on a condition.
DEFAULT	Assigns a default value when none is provided.

LAB EXERCISES:

Lab 1: Create a table courses with columns: course_id, course_name, and course_credits. Set the course_id as the primary key.

→

```
CREATE TABLE courses (
  course_id INT PRIMARY KEY AUTO_INCREMENT,
  course_name VARCHAR(100) NOT NULL,
  course_credits INT CHECK (course_credits > 0)
);
```

Lab 2: Use the CREATE command to create a database university_db.

→

```
CREATE DATABASE university_db;
```

4. Data Manipulation Language (DML)

Theory Questions:

1. Define the INSERT, UPDATE, and DELETE commands in SQL.

→

A) INSERT
- The **INSERT** command is used to add new records into a table.
- Syntax :

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

- Example :

```
INSERT INTO students (student_id, student_name, age, class)
VALUES (1, 'John Doe', 12, '6th Grade');
```

B) B. UPDATE
- The **UPDATE** command is used to modify existing records in a table.
- Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

- Example :

```
UPDATE students
SET age = 13
WHERE student_id = 1;
```

C). DELETE
- The **DELETE** command is used to remove records from a table.
- Syntax

```
DELETE FROM table_name WHERE condition;
```

- Example

```
DELETE FROM students WHERE student_id = 1;
```

2. What is the importance of the WHERE clause in UPDATE and DELETE operations?

→ The **WHERE** clause is **critical** in UPDATE and DELETE operations to **target specific records**.

→ WHERE Clause Important?
1) Prevents Unintended Changes
2) Avoids Accidental Data Loss
3) Improves Query Efficiency

→ Safe UPDATE and DELETE Examples Using WHERE :

```
UPDATE students SET age = 14 WHERE student_id = 2;
DELETE FROM students WHERE student_id = 3;
```

LAB EXERCISES:

Lab 1: Insert three records into the courses table using the INSERT command.

→

```
INSERT INTO courses (course_id, course_name, course_credits, course_duration)
VALUES
(1, 'Mathematics', 4, 6),
(2, 'Physics', 3, 5),
(3, 'Computer Science', 4, 8);
```

Lab 2: Update the course duration of a specific course using the UPDATE command

→

```
UPDATE courses
SET course_duration = 7
WHERE course_id = 2;
```

Lab 3: Delete a course with a specific course_id from the courses table using the DELETE command.

→

```
DELETE FROM courses
WHERE course_id = 3;
```

```
SELECT * FROM courses;
```

5. Data Query Language (DQL)

Theory Questions:

1. What is the SELECT statement, and how is it used to query data?

- The **SELECT** statement is used in SQL to **retrieve data** from one or more tables. It allows filtering, sorting, and formatting results based on specific requirements.

- Syntax :
  SELECT column1, column2 FROM table_name;

- column1, column2: The specific columns to retrieve.
- table_name: The table from which data is fetched.
- Example: Retrieve All Students
  SELECT * FROM students;
- This fetches all records from the students table.

2. Explain the use of the ORDER BY and WHERE clauses in SQL queries ?
->
A. WHERE Clause (Filtering Data)
- The WHERE clause is used to filter records based on a condition.
- Syntax :
  SELECT column1, column2 FROM table_name WHERE condition;
- EX
  SELECT * FROM students WHERE age > 10;

B. ORDER BY Clause (Sorting Data)
- The ORDER BY clause sorts query results in ascending (ASC) or descending (DESC) order.
- Syntax:
  SELECT column1, column2 FROM table_name ORDER BY column_name ASC|DESC;
- EX
  SELECT student_name, age FROM students ORDER BY age ASC;
  SELECT student_name, age FROM students ORDER BY age DESC;

Combining WHERE and ORDER BY

```
SELECT student_name, age FROM students
WHERE age > 10
ORDER BY age DESC;
```

LAB EXERCISES:

Lab 1: Retrieve all courses from the courses table using the SELECT statement.

→

To retrieve all records from the courses table, use the SELECT statement:

→ SELECT * FROM courses;

Lab 2: : Sort the courses based on course_duration in descending order using ORDER B

→

To sort courses by course_duration from longest to shortest, use ORDER BY:

→ SELECT * FROM courses
ORDER BY course_duration DESC;

Lab 3: Limit the Results to Show Only the Top Two Courses

→

To display only the top two courses with the longest duration, use LIMIT:

→

SELECT * FROM courses
ORDER BY course_duration DESC
LIMIT 2;

6.Data Control Language (DCL)

Theory Questions:

1. What is the purpose of GRANT and REVOKE in SQL?

→ In SQL, **GRANT** and **REVOKE** are used to manage user **permissions and privileges** on a database.

GRANT:
Allows a user or role to perform specific operations (e.g., SELECT, INSERT, UPDATE, DELETE) on a database, table, or column.

REVOKE:
Removes previously assigned privileges from a user or role.

2. How Do You Manage Privileges Using GRANT and REVOKE?

→

A. Using GRANT to Assign Privileges

- Syntax

GRANT privilege(s) ON database_name.table_name TO user_name;

- Example

GRANT SELECT, INSERT ON school_db.students TO 'user1'@'localhost';

B. Using REVOKE to Remove Privileges

- Synatx

REVOKE privilege(s) ON database_name.table_name FROM user_name;

- Example

REVOKE INSERT ON school_db.students FROM 'user1'@'localhost';

--> Managing User Privileges More Securely :

1) Granting All Privileges to an Admin User

GRANT ALL PRIVILEGES ON school_db.* TO 'admin'@'localhost';

2) **Revoking All Privileges from a User**

REVOKE ALL PRIVILEGES ON school_db.* FROM 'user1'@'localhost';

LAB EXERCISES:

Lab 1: Create two new users user1 and user2 and grant user1 permission to SELECT from the courses table.

- Step 1: Create Users

```
CREATE USER 'user1'@'localhost' IDENTIFIED BY 'password1';
CREATE USER 'user2'@'localhost' IDENTIFIED BY 'password2';
```

- Step 2: Grant SELECT Permission on courses Table to user1

```
GRANT SELECT ON school_db.courses TO 'user1'@'localhost';
```

Lab 2: Revoke the INSERT permission from user1 and give it to user2.

- Step 1: Revoke INSERT from user1

```
REVOKE INSERT ON school_db.courses FROM 'user1'@'localhost';
```

- Step 2: Grant INSERT Permission to user2

```
GRANT INSERT ON school_db.courses TO 'user2'@'localhost';
```

- Verification

```
SHOW GRANTS FOR 'user1'@'localhost';
SHOW GRANTS FOR 'user2'@'localhost';
```

7.Transaction Control Language (TCL)

Theory Questions:

1. What is the purpose of the COMMIT and ROLLBACK commands in SQL?

→ In SQL, **COMMIT** and **ROLLBACK** are used to manage transactions. They ensure **data integrity** and **consistency** when making changes to a database.

-> A) commit

- Saves all changes made during the current transaction permanently in the database.
- Once committed, changes **cannot be undone**.
- Typically used after INSERT, UPDATE, or DELETE operations.
- Example

```
START TRANSACTION;
UPDATE students SET age = 15 WHERE student_id = 1;
COMMIT;
```

B) ROLLBACK

- Undoes all changes made during the current transaction.
- Used when an error occurs, preventing unwanted changes.
- Example

```
START TRANSACTION;
UPDATE students SET age = 16 WHERE student_id = 1;
ROLLBACK;
```

2.) Explain how transactions are managed in SQL databases.

→ A **transaction** is a group of SQL statements executed as a single unit. SQL databases follow the **ACID properties** to ensure reliable transactions:

A) Atomicity
Ensures all operations in a transaction are completed; otherwise, none are applied.

B) Consistency
Ensures the database remains in a valid state before and after the transaction.

C) Isolation
Prevents multiple transactions from interfering with each other.

D) Durability
Ensures committed transactions are permanently saved in the database.

-> Example: Using SAVEPOINT

```
START TRANSACTION;
INSERT INTO students VALUES (6, 'John Doe', 14, '8th', 'New York');
SAVEPOINT sp1;

INSERT INTO students VALUES (7, 'Jane Smith', 13, '7th', 'Los Angeles');
ROLLBACK TO sp1;

COMMIT;
```

LAB EXERCISES:

Lab 1: Insert a few rows into the courses table and use COMMIT to save the changes.

- Step 1: Start a Transaction

```
START TRANSACTION;
```

→ Step 2: Insert Data into courses Table

```
INSERT INTO courses (course_id, course_name, course_credits, course_duration)
VALUES (101, 'Mathematics', 4, '6 months'),
    (102, 'Physics', 3, '5 months');
```

→ Step 3: Save the Changes Using COMMIT

```
COMMIT;
```

Lab 2: Insert additional rows, then use ROLLBACK to undo the last insert operation.

- Step 1: Start a Transaction
- Step 2: Insert More Courses

```
INSERT INTO courses (course_id, course_name, course_credits, course_duration)
VALUES (103, 'Chemistry', 3, '4 months'),
    (104, 'Biology', 3, '5 months');
```

-> Step 3: Roll Back the Changes

```
ROLLBACK;
```

Lab 3: Create a SAVEPOINT before updating the courses table, and use it to roll back specific changes.

- Step 1: Start a Transaction

→ Step 2: Create a Savepoint

```
SAVEPOINT before_update;
```

-> Step 3: Update Course Duration

```
UPDATE courses SET course_duration = '8 months' WHERE course_id = 101;
UPDATE courses SET course_duration = '7 months' WHERE course_id = 102;
```

-> **Step 4: Roll Back to SAVEPOINT**

```
ROLLBACK TO before_update;
```

-> Step 5: Commit the Final Changes

```
COMMIT;
```

-> Verification

```
SELECT * FROM courses;
```

8. SQL Joins

Theory Questions:

1. Explain the concept of JOIN in SQL. What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN?

-> A **JOIN** in SQL is used to combine rows from two or more tables based on a related column between them. Joins help retrieve data that is spread across multiple tables efficiently.

-> Types of Joins and Their Differences

Join Type	Description	Example Use Case
INNER JOIN	Returns only matching rows from both tables.	Get students who have enrolled in a course.
LEFT JOIN (LEFT OUTER JOIN)	Returns all rows from the left table and matching rows from the right table. Unmatched rows from the right table return NULL.	Get all students and their enrolled courses (even if they haven't enrolled).
RIGHT JOIN (RIGHT OUTER JOIN)	Returns all rows from the right table and matching rows from the left table. Unmatched rows from the left table return NULL.	Get all courses and their enrolled students (even if no one has enrolled).
FULL OUTER JOIN	Returns all rows from both tables. If there's no match, NULL is returned.	Get all students and all courses, showing enrollments where applicable.

2. How are joins used to combine data from multiple tables?
-> Let's assume we have two tables:
-> students Table

student id	student name	age
1	alice	15
2	Bob	14
3	chaerlie	16

-> enrollments Table

student id	course id
1	101
2	102
3	103

-> Examples of Different Joins

A) INNER JOIN (Only Matching Records)

```
SELECT students.student_id, students.student_name, enrollments.course_id
FROM students
INNER JOIN enrollments ON students.student_id = enrollments.student_id;
```

B) LEFT JOIN (All Students, Even Those Without Enrollments)

```
SELECT students.student_id, students.student_name, enrollments.course_id
FROM students
LEFT JOIN enrollments ON students.student_id = enrollments.student_id;
```

C) RIGHT JOIN (All Enrollments, Even if No Student Exists)

```
SELECT students.student_id, students.student_name, enrollments.course_id
FROM students
RIGHT JOIN enrollments ON students.student_id = enrollments.student_id;
```

D) FULL OUTER JOIN (All Students and All Enrollments)

```
SELECT students.student_id, students.student_name, enrollments.course_id
FROM students
FULL OUTER JOIN enrollments ON students.student_id = enrollments.student_id;
```

LAB EXERCISES:

Lab 1: Create two tables: departments and employees. Perform an INNER JOIN to display employees along with their respective departments

Step 1: Create departments Table

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);
```

Step 2: Insert Data into departments

```
INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'HR'),
(2, 'IT'),
(3, 'Finance');
```

Step 3: Create employees Table

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

Step 4: Insert Data into employees

```
INSERT INTO employees (emp_id, emp_name, dept_id) VALUES
(101, 'Alice', 1),
(102, 'Bob', 2),
(103, 'Charlie', 2),
(104, 'David', 3);
```

Step 5: Perform INNER JOIN to Display Employees with Their Departments

```
SELECT employees.emp_id, employees.emp_name, departments.dept_name
FROM employees
INNER JOIN departments ON employees.dept_id = departments.dept_id;
```

Lab 2: Use a LEFT JOIN to show all departments, even those without employees.

-> Step 1: Insert a Department Without Employees

```
INSERT INTO departments (dept_id, dept_name) VALUES (4, 'Marketing');
```

-> Step 2: Perform LEFT JOIN

```
SELECT departments.dept_id, departments.dept_name, employees.emp_name
FROM departments
LEFT JOIN employees ON departments.dept_id = employees.dept_id;
```

9.SQL Group By

Theory Questions

1. What is the GROUP BY clause in SQL? How is it used with aggregate functions?

-> The **GROUP BY** clause is used to **group rows that have the same values** in specified columns. It is typically used with **aggregate functions** like:

COUNT() – Counts rows in each group
SUM() – Calculates the total sum of a column for each group
AVG() – Computes the average value of a column for each group
MAX() / MIN() – Finds the highest or lowest value in a group

-> **Example:**

```
SELECT dept_id, COUNT(emp_id) AS total_employees
FROM employees
GROUP BY dept_id;
```

2. Difference Between GROUP BY and ORDER BY

->

Feature	GROUP BY	ORDER BY
Purpose	Groups rows into **categories**	Sorts the result set in **ascending (ASC) or descending (DESC) order**
Used With	Aggregate functions (COUNT(), SUM(), AVG())	Any column or expression
Result	Fewer rows (one per group)	Same number of rows but in a sorted order
Example Query	SELECT dept_id, COUNT(*) FROM employees GROUP BY dept_id;	SELECT emp_name, age FROM employees ORDER BY age DESC;

-> Example show botth

```
SELECT dept_id, COUNT(emp_id) AS total_employees
FROM employees
GROUP BY dept_id
ORDER BY total_employees DESC;
```

LAB EXERCISES:

Lab 1: Group employees by department and count the number of employees in each department using GROUP BY.

→ Step 1: Create employees Table

```
CREATE TABLE employees (
emp_id INT PRIMARY KEY,
emp_name VARCHAR(50) NOT NULL,
dept_id INT,
salary DECIMAL(10,2),
FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

-> Step 2: Insert Data into employees

```
INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES
(101, 'Alice', 1, 50000),
(102, 'Bob', 2, 60000),
(103, 'Charlie', 2, 70000),
(104, 'David', 3, 55000),
(105, 'Eve', 1, 52000);
```

-> Step 3: Group Employees by Department & Count Employees

```
SELECT dept_id, COUNT(emp_id) AS total_employees
FROM employees
GROUP BY dept_id;
```

Lab 2: Use the AVG aggregate function to find the average salary of employees in each department.

```
-> SELECT dept_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept_id;
```

10.SQL Stored Procedure

Theory Questions:

1. What is a stored procedure in SQL, and how does it differ from a standard SQL query?

-> A **stored procedure** is a precompiled collection of **one or more SQL statements** that are stored in the database and can be executed as a single unit.

-> Difference Between a Stored Procedure and a Standard SQL Query.

Feature	Stored Procedure	Standard SQL Query
Definition	A named SQL script stored in the database	A single SQL statement executed at runtime
Execution	Can be executed multiple times with different parameters	Runs once per execution
Performance	Precompiled and optimized for faster execution	Needs to be parsed and optimized each time it runs
Modularity	Can contain multiple SQL statements and logic	Usually a single SQL statement
Security	Access can be restricted using user permissions	More vulnerable to unauthorized execution

-> EXAMPLE :

Standard SQL Query

- SELECT emp_name, salary FROM employees WHERE dept_id = 1;

Stored Procedure

- CREATE PROCEDURE GetEmployeesByDept(IN dept INT)
  BEGIN
  SELECT emp_name, salary FROM employees WHERE dept_id = dept;
  END;

2. Explain the advantages of using stored procedures.

-> A) **Performance Optimization**

- Stored procedures **reduce execution time** by being precompiled.
- Less network traffic since multiple SQL queries can run within one procedure.

B) **Reusability & Maintainability**
- > Once created, stored procedures can be used **multiple times** without rewriting queries.
-> If the logic changes, updating the stored procedure **updates all calls** to it.

C) Security & Access Control
-> Permissions can be granted on stored procedures, **hiding direct access** to tables.
-> Prevents SQL injection when used with **parameters**.

D) Encapsulation of Business Logic
-> Complex logic (loops, conditions) can be **written inside procedures** rather than application code.

E) Transaction Management
-> Stored procedures **support transactions** (COMMIT, ROLLBACK) to ensure data integrity.

LAB EXERCISES:

Lab 1: Write a stored procedure to retrieve all employees from the employees table based on department.

-> Step 1: Create the Stored Procedure

```
DELIMITER $$
CREATE PROCEDURE GetEmployeesByDept(IN dept INT)
BEGIN
   SELECT emp_id, emp_name, salary
   FROM employees
   WHERE dept_id = dept;
END $$
DELIMITER ;
```

Step 2: Execute the Stored Procedure

```
-> CALL GetEmployeesByDept(1);
```

Lab 2: Write a stored procedure that accepts course_id as input and returns the course details.

-> Step 1: Create the Stored Procedure

```
DELIMITER $$
CREATE PROCEDURE GetCourseDetails(IN courseID INT)
BEGIN
   SELECT * FROM courses WHERE course_id = courseID;
END $$
DELIMITER ;
```

-> Step 2: Execute the Stored Procedure

```
CALL GetCourseDetails(101);
```

11. SQL View

Theory Questions:

1. What is a view in SQL, and how is it different from a table?

-> A **view** in SQL is a **virtual table** that is created using the result of a SELECT query. Unlike a physical table, a view **does not store data** but dynamically presents data from underlying tables whenever queried.

-> Difference Between a View and a Table

Feature	View	Table
Definition	A virtual table based on a SQL query	A physical storage unit in the database
Data Storage	Does **not** store data physically	Stores data physically
Updation	Can be updatable (in some cases)	Can be modified directly
Performance	Faster for **read operations** but slower for updates	Efficient for all CRUD operations
Security	Can **restrict access** to specific columns or rows	Full access unless restricted
Example	CREATE VIEW student_view AS SELECT student_id, student_name FROM students;	CREATE TABLE students (student_id INT, student_name VARCHAR(50));

2. Explain the advantages of using viewsin SQL databases.

-> A) **Enhanced Data Security**

- **Views can hide sensitive columns** from users by displaying only necessary data.
- EXAMPLE

```
CREATE VIEW student_info AS
SELECT student_id, student_name, age
FROM students;
```

B) **Simplifies Complex Queries**

-> Instead of writing long queries repeatedly, **create a view once and reuse it.**

-> Example

```
CREATE VIEW employee_salary AS
SELECT emp_name, salary FROM employees WHERE salary > 50000;
```

C) Improves Maintainability & Code Reusability
-> If the table structure changes, **updating the view** reflects changes in all queries using the view.

D) **Logical Data Independence**
-> Views act as an **interface** between users and tables.
-> Underlying table structure can change without affecting queries using views.

E) Performance Optimization for Read-Heavy Operations
-> A **materialized view** stores results **physically** for faster access.

LAB EXERCISES:

Lab 1: Create a view to show all employees along with their department names

-> Step 1: Create the employees and departments Tables

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    salary DECIMAL(10,2),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

-> Step 2: Insert Sample Data

```
INSERT INTO departments VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');

INSERT INTO employees VALUES
(101, 'Alice', 60000, 1),
(102, 'Bob', 45000, 2),
(103, 'Charlie', 70000, 3),
(104, 'David', 48000, 1);
```

-> Step 3: Create the View

```
CREATE VIEW Employee_Department AS
SELECT e.emp_id, e.emp_name, e.salary, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
```

-> Step 4: Retrieve Data Using the View

```
SELECT * FROM Employee_Department;
```

Lab 2: Modify the view to exclude employees whose salaries are below $50,000.

-> **Step 1: Drop the Existing View**

```
DROP VIEW IF EXISTS Employee_Department;
```

-> Step 2: Create the Modified View

```
CREATE VIEW Employee_Department AS
SELECT e.emp_id, e.emp_name, e.salary, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.salary >= 50000;
```

-> Step 3: Retrieve Data Using the Modified View

```
SELECT * FROM Employee_Department;
```

12. SQL Triggers

Theory Questions:

1. What is a trigger in SQL? Describe its types and when they are used.

-> A **trigger** in SQL is a special type of stored procedure that automatically executes in response to specific events on a table or view, such as INSERT, UPDATE, or DELETE. Triggers help enforce business rules, maintain data integrity, and automate system tasks.

-> **Types of Triggers in SQL**

SQL triggers can be classified based on their **execution timing** and **events**:

A. Based on Execution Timing

1. **Before Trigger**
   - Executes before the event (INSERT, UPDATE, or DELETE) occurs.
   - Used for validation, modifying input values before insertion, or checking constraints.
2. **After Trigger (Post-Trigger)**
   - Executes after the event occurs.
   - Commonly used for logging changes or updating related tables.
3. **Instead Of Trigger**
   - Executes **instead of** the triggering event (only applicable to views).
   - Used to modify complex views that do not support direct INSERT, UPDATE, or DELETE.

B. Based on Events

1. INSERT Trigger

- Fires when a new record is inserted into a table.
- Used for audit logging, assigning default values, or checking business rules.

2. UPDATE Trigger

- Fires when an existing record is updated.
- Used to track modifications, enforce security, or prevent accidental updates.

3. DELETE Trigger

- Fires when a record is deleted from a table.
- Used to maintain data integrity, prevent unauthorized deletions, or store deleted data in an archive table.

1. ## 2. Difference Between INSERT, UPDATE, and DELETE Triggers

Feature	INSERT Trigger	UPDATE Trigger	DELETE Trigger
When it Fires	When a new row is inserted	When an existing row is updated	When a row is deleted
Purpose	Validate or modify inserted data, log insertions	Track changes, enforce update rules	Prevent accidental deletions, log deletions
Usage Scenario	Assign default values, insert timestamp, validate I nput	Maintain historical data, restrict updates	Archive deleted records, enforce referential integrity
Example Use Case	Automatically insert a creation timestamp	Track last modified date	Move deleted records to an audit table

LAB EXERCISES:

Lab 1: Create a trigger to automatically log changes to the employees table when a new employee is added.

-> Step 1: Create a Log Table

```
CREATE TABLE employee_log (
  log_id INT AUTO_INCREMENT PRIMARY KEY,
  employee_id INT,
  name VARCHAR(255),
  action VARCHAR(50),
  action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

-> Step 2: Create an AFTER INSERT Trigger

```
CREATE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employee_log (employee_id, name, action)
  VALUES (NEW.id, NEW.name, 'INSERT');
END;
//
DELIMITER ;
```

-> Testing the Triggers

```
 INSERT INTO employees (id, name) VALUES (1, 'John Doe');
SELECT * FROM employee_log;
```

Lab 2: Create a Trigger to Update the last_modified Timestamp

-> Step 1: Modify the employees Table

```
ALTER TABLE employees ADD COLUMN last_modified
TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP;
```

-> Step 2: Create an AFTER UPDATE Trigger

```
DELIMITER //
CREATE TRIGGER before_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    SET NEW.last_modified = CURRENT_TIMESTAMP;
END;
//
DELIMITER ;
```

-> Testing the Triggers

```
UPDATE employees SET name = 'Jane Doe' WHERE id = 1;
SELECT last_modified FROM employees WHERE id = 1;
```

13. Introduction to PL/SQL

Theory Questions:

1. What is PL/SQL, and how does it extend SQL's capabilities?

-> PL/SQL (Procedural Language for SQL) is Oracle's procedural extension of SQL that allows writing procedural logic inside the database. It adds programming constructs like **loops, conditional statements, exception handling, and stored procedures** to SQL, making it more powerful and efficient for database management.

-> **How PL/SQL Extends SQL's Capabilities**

- **Procedural Programming**: Adds control structures (IF-THEN-ELSE, LOOP, WHILE, FOR) to SQL.
- **Modularity**: Supports functions, procedures, and packages for code reusability.
- **Error Handling**: Provides exception handling to manage runtime errors.
- **Performance Optimization**: Executes multiple SQL statements as a single block, reducing network traffic.
- **Triggers & Cursors**: Allows defining triggers and using cursors for row-by-row processing

2. List and explain the benefits of using PL/SQL.

-> 1. Improved Performance

- Reduces network round trips by executing multiple SQL statements in a single block.

2. Modularity

- Allows the use of procedures, functions, and packages for reusable code.

3. Better Error Handling

- Provides robust exception handling with EXCEPTION blocks to manage errors.

4. Security

- Stored procedures and functions help restrict direct database access, ensuring security.

5. Maintainability

- Code can be structured using modular design, making it easier to debug and maintain.

6. Portability

- PL/SQL code runs on any Oracle database, ensuring consistency across different environments.

7. Tight Integration with SQL

- PL/SQL fully supports SQL, allowing complex queries, DML operations, and transaction control.

LAB EXERCISES:

Lab 1: Write a PL/SQL block to print the total number of employees from the employees table.

-> This block retrieves and prints the total number of employees from the employees table.

-> PL/SQL Code:

```
DECLARE
   v_total_employees NUMBER; -- Variable to store the count
BEGIN
   -- Count the total number of employees
   SELECT COUNT(*) INTO v_total_employees FROM employees;
   -- Print the result
   DBMS_OUTPUT.PUT_LINE('Total number of employees: ' || v_total_employees);
END;
/
```

-> Executing the Block:

```
SET SERVEROUTPUT ON;
```

-> **Expected Output**

(Employee Count): Total number of employees: 150

Lab 2: Create a PL/SQL block that calculates the total sales from an orders table.

-> This block calculates and prints the total sales amount from the orders table.

-> PL/SQL Code:

```
DECLARE
   v_total_sales NUMBER := 0; -- Variable to store total sales
BEGIN
   -- Calculate total sales
   SELECT SUM(order_amount) INTO v_total_sales FROM orders;

   -- Print the result
   DBMS_OUTPUT.PUT_LINE('Total Sales Amount: ' || v_total_sales);
END;
/
```

-> **Expected Output**

Total Sales Calculation Total Sales Amount: 50000

14. PL/SQL Control Structures

Theory Questions:

1. What are control structures in PL/SQL? Explain the IF-THEN and LOOP control structures.

-> Control structures in **PL/SQL** are used to control the flow of execution in a program. They allow conditional execution, iteration, and branching, making PL/SQL more powerful than standard SQL.

-> **Types of Control Structures in PL/SQL:**

1. **Conditional Control (IF-THEN, CASE)**
2. **Looping Control (LOOP, WHILE, FOR)**
3. **Sequential Control (GOTO, NULL)**

1 . IF-THEN Control Structure

-> The IF-THEN structure is used for **conditional execution**.

-> Syntax

```
 IF condition THEN
    -- Statements to execute if the condition is TRUE
ELSIF another_condition THEN
    -- Statements if another condition is TRUE
ELSE
    -- Statements if none of the conditions are TRUE
END IF;
```

-> Example: Checking Employee Salary

```
DECLARE
    v_salary NUMBER := 5000;
BEGIN
    IF v_salary > 10000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary');
    ELSIF v_salary BETWEEN 5000 AND 10000 THEN
        DBMS_OUTPUT.PUT_LINE('Medium Salary');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Low Salary');
    END IF;
END;
/
```

2. LOOP Control Structure

-> Loops are used to execute a block of code repeatedly until a certain condition is met.

-> **Types of Loops:**
BASIC LOOP (Requires EXIT condition inside)
WHILE LOOP (Executes while condition is TRUE)
FOR LOOP (Runs a fixed number of times)

-> 1. BASIC LOOP Example

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 5; -- Condition to exit loop
  END LOOP;
END;
/
```

-> 2. WHILE LOOP Example

```
DECLARE
  v_num NUMBER := 1;
BEGIN
  WHILE v_num <= 5 LOOP
    DBMS_OUTPUT.PUT_LINE('Number: ' || v_num);
    v_num := v_num + 1;
  END LOOP;
END;
/
```

3. FOR LOOP Example

```
-> BEGIN
  FOR i IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration: ' || i);
  END LOOP;
END;
/
```

LAB EXERCISES:

1. Lab 1: Write a PL/SQL block using an IF-THEN condition to check the department of an employee.

-> PL/SQL Code:

```
DECLARE
   v_emp_id NUMBER := 101;  -- Change this to any employee ID
   v_dept VARCHAR2(50);
BEGIN
   -- Fetch the department of the employee
   SELECT department_name INTO v_dept
   FROM employees
   WHERE employee_id = v_emp_id;

   -- Check department and print a message
   IF v_dept = 'HR' THEN
      DBMS_OUTPUT.PUT_LINE('Employee is in Human Resources.');
   ELSIF v_dept = 'IT' THEN
      DBMS_OUTPUT.PUT_LINE('Employee is in the IT Department.');
   ELSIF v_dept = 'Finance' THEN
      DBMS_OUTPUT.PUT_LINE('Employee is in Finance.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee is in another department.');
   END IF;
END;
/
```

-> Expected Output:

Employee is in the IT Department.

Lab 2: PL/SQL Block Using FOR LOOP to Display Employee Names

-> PL/SQL Code:

```
DECLARE
   CURSOR emp_cursor IS SELECT employee_name
FROM employees;
   v_name employees.employee_name%TYPE;
BEGIN
   FOR emp_rec IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Employee Name: ' ||
emp_rec.employee_name);
   END LOOP;
END;
/
```

-> Expected Output:

```
Employee Name: John Doe
Employee Name: Jane Smith
Employee Name: Michael Brown
...
```

15. SQL Cursors

Theory Questions:

1. What is a cursor in PL/SQL? Explain the difference between implicit and explicit cursors.

-> A **cursor** in PL/SQL is a pointer to a result set of a query. It is used to **fetch and process multiple rows** returned by a SQL statement.

->PL/SQL **automatically creates** a cursor for each SQL query, but we can also **define our own** cursors for more control.

-> Difference Between Implicit and Explicit Cursors

Feature	Implicit Cursor	Explicit Cursor
Definition	Automatically created by PL/SQL for SELECT, INSERT, UPDATE, or DELETE statements.	Defined manually by the user for queries that return multiple rows.
Control	No control over row-by-row processing.	Allows row-by-row processing with OPEN, FETCH, and CLOSE.
Performance	Faster and optimized for single-row operations.	Useful when handling multiple rows efficiently.
usage	Used when a query returns **only one row**.	Used when a query returns **multiple rows**.
example	SELECT COUNT(*) INTO v_count FROM employees;	CURSOR emp_cursor IS SELECT * FROM employees;

-> Example of an Implicit Cursor

```
DECLARE
   v_total NUMBER;
BEGIN
   SELECT COUNT(*) INTO v_total FROM employees;
   DBMS_OUTPUT.PUT_LINE('Total Employees: ' || v_total);
END;
/
```

2. When Would You Use an Explicit Cursor Over an Implicit One?

-> **Processing multiple rows**
– If a query returns more than one row, explicit cursors allow fetching each row one by one.
-> Complex business logic
– When you need conditional processing on each row.
-> Performance optimization
– If you need to control how and when rows are fetched.
-> Fetching multiple columns
– If your query retrieves multiple columns and you want to process them row by row.

-> EXAMPLE

```
DECLARE
   CURSOR emp_cursor IS SELECT employee_id, salary FROM employees;
   v_id employees.employee_id%TYPE;
   v_salary employees.salary%TYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO v_id, v_salary;
      EXIT WHEN emp_cursor%NOTFOUND;

      IF v_salary > 5000 THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || v_id || ' has a high salary.');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Employee ' || v_id || ' has a normal salary.');
      END IF;
   END LOOP;
   CLOSE emp_cursor;
END;
/
```

LAB EXERCISES:

- Lab 1: Write a PL/SQL block using an explicit cursor to retrieve and display employee details.

-> PL/SQL Code:

```
DECLARE
   -- Declare an explicit cursor to fetch employee details
   CURSOR emp_cursor IS
      SELECT employee_id, employee_name, salary, department_name
      FROM employees;

   -- Declare variables to store fetched values
   v_id employees.employee_id%TYPE;
   v_name employees.employee_name%TYPE;
   v_salary employees.salary%TYPE;
   v_dept employees.department_name%TYPE;
BEGIN
   -- Open the cursor
   OPEN emp_cursor;

   LOOP
      -- Fetch data from the cursor into variables
      FETCH emp_cursor INTO v_id, v_name, v_salary, v_dept;

      -- Exit loop when no more records are found
      EXIT WHEN emp_cursor%NOTFOUND;

      -- Display employee details
      DBMS_OUTPUT.PUT_LINE('ID: ' || v_id || ', Name: ' || v_name ||
                           ', Salary: ' || v_salary || ', Dept: ' || v_dept);
   END LOOP;

   -- Close the cursor
   CLOSE emp_cursor;
END;
/
```

-> Expected Output for Employess

```
ID: 101, Name: John Doe, Salary: 5000, Dept: IT
ID: 102, Name: Jane Smith, Salary: 6000, Dept: HR
...
```

Lab 2: Create a cursor to retrieve all courses and display them one by one.

-> PL/SQL Code:

```
DECLARE
   -- Declare an explicit cursor to fetch course details
   CURSOR course_cursor IS
      SELECT course_id, course_name FROM courses;

   -- Declare variables to store fetched values
   v_course_id courses.course_id%TYPE;
   v_course_name courses.course_name%TYPE;
BEGIN
   -- Open the cursor
   OPEN course_cursor;

   LOOP
      -- Fetch data from the cursor into variables
      FETCH course_cursor INTO v_course_id, v_course_name;

      -- Exit loop when no more records are found
      EXIT WHEN course_cursor%NOTFOUND;

      -- Display course details
      DBMS_OUTPUT.PUT_LINE('Course ID: ' || v_course_id || ', Name: ' ||
v_course_name);
   END LOOP;

   -- Close the cursor
   CLOSE course_cursor;
END;
/
```

-> Expected Output for Course

Course ID: 1, Name: Database Management
Course ID: 2, Name: Web Development
...

16. Rollback and Commit Savepoint

Theory Questions:

1. Explain the concept of SAVEPOINT in transaction management. How do ROLLBACK and COMMIT interact with savepoints?

-> A **SAVEPOINT** in SQL is a marker within a transaction that allows partial rollbacks. It is used to **split** a large transaction into smaller parts, so if an error occurs, only a portion of the transaction is rolled back instead of the entire transaction.

- -> **Interaction of SAVEPOINT with COMMIT and ROLLBACK**

  A) COMMIT:
  Finalizes the transaction, making all changes **permanent** in the database.

  B) ROLLBACK TO SAVEPOINT: Reverts changes **only up to a specific savepoint**, keeping the earlier operations intact.

  C) ROLLBACK (without savepoint): Undoes **all changes** in the transaction.

-> Example of SAVEPOINT, COMMIT, and ROLLBACK

```
-- Start a transaction
BEGIN;

-- Insert a record into Employees
INSERT INTO employees (employee_id, employee_name, salary) VALUES (101,
    'John Doe', 5000);

-- Create a savepoint
SAVEPOINT emp_save;

-- Insert another record
INSERT INTO employees (employee_id, employee_name, salary) VALUES (102,
    'Jane Smith', 6000);

-- Oops! Assume something goes wrong, rollback to savepoint
ROLLBACK TO emp_save;

-- Now, only the first INSERT remains, the second is undone
COMMIT;
```

2. When is it Useful to Use Savepoints?

->

1) Handling Errors in Complex Transactions

-> If a transaction involves **multiple dependent operations**, using savepoints prevents rolling back the entire transaction if only part of it fails.

-> Example: In a banking system, transferring money between accounts.

2) Testing and Debugging Transactions

-> Developers can use savepoints to check intermediate results without rolling back the full transaction.

3) **Batch Processing Large Transactions**

-> Inserting/updating thousands of records, but only rolling back a specific section instead of the entire batch.

4) **Ensuring Data Integrity with Conditional Checks**

-> If some conditions fail after processing part of a transaction, only the failed part is rolled back.

-> Example: Using SAVEPOINT for Order Processing

```
BEGIN;

-- Step 1: Insert Order Details
INSERT INTO orders (order_id, customer_name, total_amount) VALUES (5001, 'Alice', 100);

-- Create Savepoint
SAVEPOINT order_step1;

-- Step 2: Deduct from Inventory
UPDATE inventory SET stock = stock - 1 WHERE product_id = 2001;

-- Something goes wrong (out of stock), rollback to savepoint
ROLLBACK TO order_step1;

-- Order remains, but inventory change is undone
COMMIT;
```

LAB EXERCISES:

Lab 1: Perform a transaction where you create a savepoint, insert records, then rollback to the savepoint.

-> **PL/SQL Code**

```
BEGIN
   -- Insert the first employee record
   INSERT INTO employees (employee_id, employee_name, salary)
     VALUES (201, 'Alice', 7000);

   -- Create a savepoint
   SAVEPOINT emp_save;

   -- Insert another record
   INSERT INTO employees (employee_id, employee_name, salary)
     VALUES (202, 'Bob', 6500);

   -- Oops! Assume some error occurred, rollback to savepoint
   ROLLBACK TO emp_save;

   -- Commit the transaction (Only Alice's record is saved)
   COMMIT;

   DBMS_OUTPUT.PUT_LINE('Transaction committed after rollback
     to savepoint.');
END;
/
```

Lab 2: Commit part of a transaction after using a savepoint and then rollback the remaining changes.

-> PL/SQL Code

```
BEGIN
   -- Insert first record
   INSERT INTO employees (employee_id, employee_name, salary)
VALUES (301, 'Charlie', 7200);

   -- Create a savepoint
   SAVEPOINT emp_save1;

   -- Insert second record
   INSERT INTO employees (employee_id, employee_name, salary)
VALUES (302, 'David', 6800);

   -- Commit the first part (Charlie's record is saved)
   COMMIT;

   -- Create another savepoint after commit
   SAVEPOINT emp_save2;

   -- Insert third record
   INSERT INTO employees (employee_id, employee_name, salary)
VALUES (303, 'Eve', 7500);

   -- Rollback the remaining transaction (Only Eve's record is rolled back)
   ROLLBACK TO emp_save2;

   DBMS_OUTPUT.PUT_LINE('First part committed, second part rolled
back.');
END;
/
```