<u>Experiment – 1</u>

Create a Database Schema for University Database

Aim: To create a University Database with tables for Students, Courses, and Enrollments, and to demonstrate relationships and queries between them.

Algorithm/ Procedure:

1. Create a database `UniversityDB`.
2. Create tables: `Student`, `Course`, and `Enrollment` with appropriate keys.
3. Insert sample data into all tables.
4. Execute queries to display students, courses, enrollments, and their relationships.

Program:

1. Create Database and Use It

```
CREATE DATABASE UniversityDB; USE

UniversityDB;
```

2. Create Tables

```
-- Student Table

CREATE TABLE Student (

   student_id INT PRIMARY KEY AUTO_INCREMENT, name

   VARCHAR(50) NOT NULL,

   age INT,

   department VARCHAR(50) );

-- Course Table   CREATE

TABLE Course (

   course_id INT PRIMARY KEY AUTO_INCREMENT,

   course_name VARCHAR(50) NOT NULL,

   credits INT

);
```

-- Enrollment Table (Many-to-Many Relationship) CREATE

TABLE Enrollment (

enroll_id INT PRIMARY KEY AUTO_INCREMENT,

student_id INT,

course_id INT,

FOREIGN KEY (student_id

Output:

1. Show all students

	student_id	name	age	department
▶	1	Alice	20	Computer Science
	2	Bob	21	Mathematics
	3	Charlie	22	Physics
*	NULL	NULL	NULL	NULL

2. **Show all courses**

	course_id	course_name	credits
▶	1	Database Systems	4
	2	Calculus	3
	3	Quantum Mechanics	4
*	NULL	NULL	NULL

3. Show all enrollments

	enroll_id	student_id	course_id
▶	1	1	1
	2	1	2
	3	2	2
	4	3	3
*	NULL	NULL	NULL

Result: **The program executed successfully. Tables were created, data inserted, and queries ran correctly, displaying the expected results.**

Experiment – 2

SQL queries for employee database with key constraints

Aim: To create an Employee Database with tables for Employees and Departments, manage their records, and perform basic queries like display, update, and delete.

Algorithm/ Procedure:

1. Create a database `EmployeeDB`.
2. Create `Department` and `Employee` tables with primary key, foreign key, and constraints.
3. Insert sample data into both tables.
4. Execute queries to display all employees, join with departments, filter by salary, count employees per department, update salary, and delete a record.

Program:

```
-- 1. Create Database and Use It
CREATE DATABASE EmployeeDB;
USE EmployeeDB;

-- 2. Create Tables

-- Department Table CREATE TABLE
Department (
    dept_id INT PRIMARY KEY AUTO_INCREMENT, dept_name
    VARCHAR(50) UNIQUE NOT NULL
);

-- Employee Table CREATE
TABLE Employee (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,                    -- Primary Key
    emp_name VARCHAR(50) NOT NULL,                            -- Not Null
    phone_number VARCHAR(15) UNIQUE,                         --Unique
    address VARCHAR(100),
    salary DECIMAL(10,2) CHECK (salary > 0),                 -- Check constraint dept_id
    INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)   -- Foreign
Key
);
```

```sql
-- 3. Insert Records

-- Insert Departments
INSERT INTO Department (dept_name) VALUES

('HR'),
('IT'),
('Finance');

-- Insert Employees
INSERT INTO Employee (emp_name, phone_number, address, salary, dept_id) VALUES
('Alice Johnson', '9876543210', 'Delhi', 55000, 1),
('Bob Smith', '9876500000', 'Mumbai', 72000, 2),
('Charlie Brown', '9876511111', 'Kolkata', 60000, 3),
('Diana Prince', '9876522222', 'Chennai', 80000, 2);

-- 4. Queries / Operations

-- 4.1 Show all employees SELECT *
FROM Employee;

-- 4.2 List employees with their department names SELECT e.emp_name,
e.phone_number, e.salary, d.dept_name FROM Employee e
JOIN Department d ON e.dept_id = d.dept_id;

-- 4.3 Find employees earning more than ₹60,000 SELECT
emp_name, salary
FROM Employee
WHERE salary > 60000;

-- 4.4 Count number of employees in each department SELECT d.dept_name,
COUNT(e.emp_id) AS total_employees FROM Department d
LEFT JOIN Employee e ON d.dept_id = e.dept_id GROUP BY
d.dept_name;

-- 4.5 Update salary for an employee UPDATE Employee
SET salary = salary + 5000
WHERE emp_name = 'Alice Johnson';

-- 4.6 Delete an employee
DELETE FROM Employee
WHERE emp_id = 3;
```

Output:

1. Show all employees

	emp_id	emp_name	phone_number	address	salary	dept_id
▶	1	Alice Johnson	9876543210	Delhi	55000.00	1
	2	Bob Smith	9876500000	Mumbai	72000.00	2
	3	Charlie Brown	9876511111	Kolkata	60000.00	3
	4	Diana Prince	9876522222	Chennai	80000.00	2
*	NULL	NULL	NULL	NULL	NULL	NULL

2. List employees with their department names

	emp_name	phone_number	salary	dept_name
▶	Charlie Brown	9876511111	60000.00	Finance
	Alice Johnson	9876543210	55000.00	HR
	Bob Smith	9876500000	72000.00	IT
	Diana Prince	9876522222	80000.00	IT

3. Employees earning more than ₹60,000

	emp_name	salary
▶	Bob Smith	72000.00
	Diana Prince	80000.00

4. Count number of employees in each department

	dept_name	total_employees
▶	Finance	1
	HR	1
	IT	2

Result: The program executed successfully. Tables were created, data inserted correctly, and queries returned the expected results

Experiment – 3

Create ER Model University Database

Aim: To create a University Database, perform table operations like adding columns, renaming, truncating, and dropping a table.

Algorithm/ Procedure:

1. Create a database `UniversityDB`.
2. Create a `Student` table with columns for ID, name, age, and department.
3. Alter the table to add an `email` column.
4. Rename the table from `Student` to `Students`.
5. Truncate the table to remove all data.
6. Drop the table to delete it completely.

Program:

```
-- 1. Create Database
CREATE DATABASE UniversityDB; USE
UniversityDB;


-- 2. Create Tables (Entities)
-- Department Table CREATE TABLE
Department (
    dept_id INT PRIMARY KEY AUTO_INCREMENT, dept_name
    VARCHAR(50) UNIQUE NOT NULL
);

-- Student Table CREATE TABLE
Student (
    student_id INT PRIMARY KEY AUTO_INCREMENT, name
    VARCHAR(50) NOT NULL,
    age INT,
    email VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
);

-- Course Table CREATE TABLE
Course (
    course_id INT PRIMARY KEY AUTO_INCREMENT,
    course_name VARCHAR(50) NOT NULL, credits INT
);

-- Enrollment Table (Many-to-Many Relationship) CREATE TABLE
Enrollment (
```

```
        enroll_id INT PRIMARY KEY AUTO_INCREMENT,
        student_idINT, course_id INT,
        FOREIGN KEY (student_id) REFERENCES Student(student_id), FOREIGN KEY
        (course_id) REFERENCES Course(course_id)
);

-- 3. Insert Sample Data

-- Insert Departments
INSERT INTO Department (dept_name) VALUES ('Computer Science'),
('Mathematics'),
('Physics');

-- Insert Students
INSERT INTO Student (name, age, email, dept_id) VALUES ('Alice', 20,
'alice@example.com', 1),
('Bob', 21, 'bob@example.com', 2),
('Charlie', 22, 'charlie@example.com', 3),
('Diana', 19, 'diana@example.com', 1);

-- Insert Courses
INSERT INTO Course (course_name, credits) VALUES ('Database
Systems', 4),
('Calculus', 3),
('Quantum Mechanics', 4),
('Algorithms', 3);

-- Insert Enrollments
INSERT INTO Enrollment (student_id, course_id) VALUES
 (1,   1),   --   Alice → Database Systems
 (1,   4),   --   Alice → Algorithms
 (2,   2),   --   Bob → Calculus
 (3,   3),   --   Charlie → Quantum Mechanics
 (4,   1),   --   Diana → Database Systems
 (4,   4);   --   Diana → Algorithms

-- 4. Queries / Operations

-- 4.1 Show all students SELECT *
FROM Student;

-- 4.2 Show all departments SELECT * FROM
Department;

-- 4.3 Show all courses SELECT *
FROM Course;

-- 4.4 Show all enrollments SELECT * FROM
```

Enrollment;

-- 4.5 List students with their department names
SELECT s.name AS Student_Name, s.email, d.dept_name AS Department FROM Student s
JOIN Department d ON s.dept_id = d.dept_id;

-- 4.6 Which student takes which course
SELECT s.name AS Student_Name, c.course_name AS Course_Name FROM Enrollment e
JOIN Student s ON e.student_id = s.student_id JOIN Course c ON
e.course_id = c.course_id;

-- 4.7 Students in 'Computer Science' department SELECT * FROM
Student
WHERE dept_id = 1;

-- 4.8 Courses with more than 3 credits SELECT * FROM
Course
WHERE credits > 3;

Output:

1.Student table

student_id	name	age	email	dept_id
1	Alice	20	alice@example.com	1
2	Bob	21	bob@example.com	2
3	Charlie	22	charlie@example.com	3
4	Diana	19	diana@example.com	1
NULL	NULL	NULL	NULL	NULL

student_id	name	age	email	dept_id
1	Alice	20	alice@example.com	1
4	Diana	19	diana@example.com	1
NULL	NULL	NULL	NULL	NULL

2. Department table

dept_id	dept_name
1	Computer Science
2	Mathematics
3	Physics
NULL	NULL

3. Course Table

course_id	course_name	credits
1	Database Systems	4
2	Calculus	3
3	Quantum Mechanics	4
4	Algorithms	3
NULL	NULL	NULL

course_id	course_name	credits
1	Database Systems	4
3	Quantum Mechanics	4
NULL	NULL	NULL

4. Enrollement Table

enroll_id	student_id	course_id
1	1	1
2	1	4
3	2	2
4	3	3
5	4	1

5. Result Table

Student_Name	email	Department
Alice	alice@example.com	Computer Science
Diana	diana@example.com	Computer Science
Bob	bob@example.com	Mathematics
Charlie	charlie@example.com	Physics

Student_Name	Course_Name
Alice	Database Systems
Alice	Algorithms
Bob	Calculus
Charlie	Quantum Mechanics
Diana	Database Systems

<u>Result:</u> The program executed successfully. All tables were created, sample data inserted, and queries returned the expected results.

Experiment – 4

Implement DDL, DML commands

Aim: To implement DDL, DML, and DQL commands in SQL for creating, modifying, and manipulating a database and retrieving data.

Algorithm/ Procedure:

1. Create a database (`CREATE DATABASE`) and use it (`USE`).
2. Create a table (`CREATE TABLE`) with required fields.
3. Modify the table using (`ALTER TABLE`).
4. Insert records using (`INSERT`).
5. Update a record using (`UPDATE`).
6. Delete a record using (`DELETE`).
7. Retrieve data using (`SELECT`).
8. Clear table data using (`TRUNCATE`).
9. Drop the table (`DROP TABLE`).
10. Drop the database (`DROP DATABASE`).

Program:

```
-- DDL (Create and Drop Database)
CREATE DATABASE CollegeDB;
USE CollegeDB;

-- DDL (Create, Alter, Truncate, Drop Table)
CREATE TABLE Students (
   ID INT PRIMARY KEY,
   Name VARCHAR(50),
   Marks INT
);

ALTER TABLE Students ADD COLUMN Age INT;

-- DML (Insert, Update, Delete)
INSERT INTO Students (ID, Name, Marks, Age) VALUES (1, 'Rahul', 85, 20);
INSERT INTO Students (ID, Name, Marks, Age) VALUES (2, 'Priya', 92, 21);
INSERT INTO Students (ID, Name, Marks, Age) VALUES (3, 'Amit', 78, 19);

UPDATE Students SET Marks = 95 WHERE ID = 2;

DELETE FROM Students WHERE ID = 3;
```

```
-- DQL (Select Queries)
SELECT * FROM Students;                -- fetch all records
SELECT Name, Marks FROM Students;      -- fetch specific columns
SELECT * FROM Students WHERE Marks > 80; -- fetch with condition

-- DDL (Truncate and Drop Table)
TRUNCATE TABLE Students;  -- removes all rows but keeps structure
DROP TABLE Students;      -- removes table completely

-- DDL (Drop Database)
DROP DATABASE CollegeDB
```

;**<u>Output:</u>**

1. Student Table

ID	Name	Marks	Age
1	Rahul	85	20
2	Priya	95	21
NULL	NULL	NULL	NULL

Name	Marks
Rahul	85
Priya	95

ID	Name	Marks	Age
1	Rahul	85	20
2	Priya	95	21
NULL	NULL	NULL	NULL

<u>Result:</u> The SQL commands for DDL, DML, and DQL were successfully implemented and executed, demonstrating creation, modification, manipulation, retrieval, and deletion of data.

Experiment – 5

Implement DCL,TCL Commands

Aim: To implement **DCL** and **TCL** commands to manage transactions and control user privileges in SQL.

Algorithm/ Procedure:

1. Create database `BankDB` and table `Accounts`.
2. Insert sample records.
3. **TCL:** Start transaction, update balances, create savepoint, rollback to savepoint, commit.
4. **DCL:** Grant and revoke privileges to a user.
5. Verify results using `SELECT`.

Program:

```
-- CREATE DATABASE AND TABLE
CREATE DATABASE BankDB;
USE BankDB;

CREATE TABLE Accounts ( AccNo INT
    PRIMARY KEY,
    HolderNameVARCHAR(50), Balance
    DECIMAL(10,2)
);

INSERT INTO Accounts VALUES (101, 'Rahul', 5000.00); INSERT INTO Accounts
VALUES (102, 'Priya', 7000.00);

-- TCL (Transaction Control Language)

-- Start a Transaction START
TRANSACTION;

-- First Operation: Deduct 1000 from Rahul
UPDATE Accounts SET Balance = Balance - 1000 WHERE AccNo = 101;

-- Create Savepoint
SAVEPOINT BeforeSecondDeduction;

-- Second Operation: Deduct 500 from Rahul
UPDATE Accounts SET Balance = Balance - 500 WHERE AccNo = 101;

-- Rollback only the second deduction
ROLLBACK TO BeforeSecondDeduction;
```

-- Commit remaining changes COMMIT;

-- Check current balances SELECT *

FROM Accounts;

-- DCL (Data Control Language)

-- Grant privileges to a user
GRANT SELECT, INSERT, UPDATE, DELETE ON BankDB.* TO
'testuser'@'localhost';

-- Revoke privileges from a user REVOKE INSERT, DELETE
ON BankDB.* FROM

<u>Output:</u>

1. Accounts Table

AccNo	HolderName	Balance
101	Rahul	4000.00
102	Priya	7000.00
NULL	NULL	NULL

:

<u>Result:</u> Transactions executed correctly; rollback worked; privileges granted and revoked successfully; table updated as expected.

Experiment – 6

Implement SQL sub queries, Joins and Clauses

Aim: To implement SQL subqueries, joins, and clauses to retrieve and analyze data.

Algorithm/ Procedure:

1. Create database `SchoolDB` and tables `Students` and `Marks`.
2. Insert sample records into both tables.
3. Use **Joins** (INNER, LEFT, RIGHT) to combine tables.
4. Use **Subqueries** to filter students based on conditions.
5. Apply **Clauses**: `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`.
6. Verify results using `SELECT`.

Program:

```
-- CREATE DATABASE AND TABLES
CREATE DATABASE SchoolDB;
USE SchoolDB;

-- Students Table CREATE TABLE
Students (
        StudentID INT PRIMARY KEY, Name
        VARCHAR(50),
        Class VARCHAR(10)
);

-- Marks Table CREATE TABLE
Marks (
        MarkID INT PRIMARY KEY,
        StudentID INT, Subject
        VARCHAR(50), Marks INT,
        FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);

-- INSERT SAMPLE DATA
INSERT INTO Students VALUES (1, 'Rahul', '10A'); INSERT INTO Students
VALUES (2, 'Priya', '10B'); INSERT INTO Students VALUES (3, 'Amit',
'10A'); INSERT INTO Students VALUES (4, 'Sneha', '10B');
 INSERT INTO Marks VALUES (1, 1, 'Math', 85);
 INSERT INTO Marks VALUES (2, 1, 'Science', 90);
 INSERT INTO Marks VALUES (3, 2, 'Math', 78);
 INSERT INTO Marks VALUES (4, 2, 'Science', 88);
 INSERT INTO Marks VALUES (5, 3, 'Math', 92);
 INSERT INTO Marks VALUES (6, 3, 'Science', 81);
```

```sql
INSERT INTO Marks VALUES (7, 4, 'Math', 75);
INSERT INTO Marks VALUES (8, 4, 'Science', 80);

-- JOINS

-- INNER JOIN: Get student names with their marks SELECT s.Name, m.Subject,
m.Marks
FROM Students s
INNER JOIN Marks m ON s.StudentID = m.StudentID;

-- LEFT JOIN: Get all students and their marks (even if no marks) SELECT s.Name, m.Subject,
m.Marks
FROM Students s
LEFT JOIN Marks m ON s.StudentID = m.StudentID;

-- RIGHT JOIN: Get all marks and corresponding student names SELECT s.Name, m.Subject,
m.Marks
FROM Students s
RIGHT JOIN Marks m ON s.StudentID = m.StudentID;

-- SUBQUERIES
-- Find students who scored more than average in Math SELECT Name
FROM Students
WHERE StudentID IN ( SELECT
    StudentID FROM Marks
    WHERE Subject = 'Math' AND Marks > (
        SELECT AVG(Marks) FROM Marks WHERE Subject = 'Math'
    )
);

-- CLAUSES

-- WHERE: Students in class 10A
SELECT * FROM Students WHERE Class = '10A';

-- ORDER BY: Students by Name
SELECT * FROM Students ORDER BY Name ASC;

-- GROUP BY: Average marks per subject SELECT Subject,
AVG(Marks) AS AvgMarks FROM Marks
GROUP BY Subject;

-- HAVING: Subjects with average marks > 80 SELECT Subject,
AVG(Marks) AS AvgMarks FROM Marks
GROUP BY Subject HAVING
AVG(Marks) > 80;
```

Output:

1. **INNER JOIN**

Name	Subject	Marks
Rahul	Math	85
Rahul	Science	90
Priya	Math	78
Priya	Science	88
Amit	Math	92
Amit	Science	81
Sneha	Math	75

2. **Subquery**

Name
Rahul
Amit

3. **WHERE Clause**

StudentID	Name	Class
1	Rahul	10A
3	Amit	10A
NULL	NULL	NULL

4. **ORDER BY Clause**

StudentID	Name	Class
3	Amit	10A
2	Priya	10B
1	Rahul	10A
4	Sneha	10B
NULL	NULL	NULL

5. **GROUP BY Clause**

Subject	AvgMarks
Math	82.5000
Science	84.7500

Result: Queries executed successfully; joins combined tables correctly; subqueries filtered data as expected; clauses sorted, grouped, and aggregated data accurately; output displayed as intended.

Experiment – 7

PL/SQL: Case, Loop.

Aim: To calculate student grades using a PL/SQL cursor and CASE statement and display roll numbers, marks, and grades.

Algorithm/ Procedure:

1. Enable server output using `SET SERVEROUTPUT ON`.
2. Declare a cursor to hold student roll numbers and marks.
3. Loop through each record of the cursor.
4. Use a `CASE` statement to assign grades based on marks.
5. Display roll number, marks, and grade using `DBMS_OUTPUT.PUT_LINE`.
6. End the loop and PL/SQL block.

Program:

```
SET SERVEROUTPUT ON;
DECLARE
  -- Cursor to simulate student marks
  CURSOR stu_cur IS
    SELECT 101 AS roll_no, 85 AS marks FROM dual
    UNION
    SELECT 102, 72 FROM dual
    UNION
    SELECT 103, 59 FROM dual
    UNION
    SELECT 104, 40 FROM dual;


  v_roll   NUMBER;
  v_marks  NUMBER;
  v_grade  CHAR(2);
BEGIN

  -- Loop through cursor


  FOR stu_rec IN stu_cur LOOP
    v_roll := stu_rec.roll_no;
    v_marks := stu_rec.marks;


    -- CASE for grade calculation
    CASE
      WHEN v_marks >= 80 THEN
        v_grade := 'A';
```

```
        WHEN v_marks >= 60 THEN
            v_grade := 'B';
        WHEN v_marks >= 50 THEN
            v_grade := 'C';
        WHEN v_marks >= 40 THEN
            v_grade := 'D';
        ELSE

            v_grade := 'F';
    END CASE;


    -- Output result
    DBMS_OUTPUT.PUT_LINE(
        'Roll No: ' || v_roll ||

        ' | Marks: ' || v_marks ||
        ' | Grade: ' || v_grade
    );

  END LOOP;

END;


/
```

<u>Output:</u>

```
Roll No: 101 | Marks: 85 | Grade: A
Roll No: 102 | Marks: 72 | Grade: B
Roll No: 103 | Marks: 59 | Grade: C
Roll No: 104 | Marks: 40 | Grade: D


PL/SQL procedure successfully completed.

Elapsed: 00:00:00.006
```

<u>Result:</u> The program executed successfully; grades were calculated correctly for all students; roll numbers, marks, and grades were displayed as expected.

Experiment – 8

Implementing PL/SQL Conditional Statements, Looping Statements

Aim: To demonstrate LOOP, FOR LOOP, WHILE LOOP, IF, and CASE statements in PL/SQL for factorial calculation, even/odd checking, and countdown.

Algorithm/ Procedure:

1. Enable server output using `SET SERVEROUTPUT ON`.
2. Use a **LOOP** with `IF` to calculate the factorial of a number.
3. Use a **FOR LOOP** with `CASE` to check and display even/odd numbers.
4. Use a **WHILE LOOP** to perform a countdown.
5. Display results using `DBMS_OUTPUT.PUT_LINE`.

Program:

```
SET SERVEROUTPUT ON;
DECLARE
  n    NUMBER := 5;     -- Input number

  fact  NUMBER := 1;     -- Variable to store factorial
  i    NUMBER := 1;     -- Counter
BEGIN

  DBMS_OUTPUT.PUT_LINE('--- Factorial Program using LOOP and IF ---');


  -- Simple LOOP to calculate factorial
  LOOP
    fact := fact * i;
    i := i + 1;


    -- Conditional check to exit loop
    IF i > n THEN
      EXIT;

    END IF;

  END LOOP;


  DBMS_OUTPUT.PUT_LINE('Factorial of ' || n || ' is ' || fact);

  DBMS_OUTPUT.PUT_LINE('--- Numbers and Even/Odd check using FOR LOOP & CASE ---');
```

```
  -- FOR LOOP + CASE to check even/odd
 FOR j IN 1..10 LOOP
    CASE

        WHEN MOD(j,2) = 0 THEN
           DBMS_OUTPUT.PUT_LINE(j || ' is EVEN');
        ELSE

           DBMS_OUTPUT.PUT_LINE(j || ' is ODD');
    END CASE;
  END LOOP;


  DBMS_OUTPUT.PUT_LINE('--- WHILE LOOP Example (Countdown) ---');


  -- WHILE LOOP for countdown
  DECLARE
     k NUMBER := 5;
  BEGIN
     WHILE k > 0 LOOP

        DBMS_OUTPUT.PUT_LINE('Countdown: ' || k);
        k := k - 1;
     END LOOP;
 END;
END;
/
```

Output:

```
--- Factorial Program using LOOP and IF ---
Factorial of 5 is 120
--- Numbers and Even/Odd check using FOR LOOP 78 ---
1 is ODD
2 is EVEN
3 is ODD
4 is EVEN
5 is ODD
6 is EVEN
7 is ODD
8 is EVEN
9 is ODD
10 is EVEN
--- WHILE LOOP Example (Countdown) ---
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
```

Result: All loops executed successfully; factorial calculated correctly; even/odd numbers displayed accurately; countdown performed as expected; output displayed properly.

Experiment – 9

Sample programs for Cursors and Exceptions

Aim: To demonstrate PL/SQL cursors and exception handling using sample programs.

Algorithm/ Procedure:

1. Cursor Program:

1. Enable server output using `SET SERVEROUTPUT ON`.
2. Declare a cursor to fetch student details from a table.
3. Loop through the cursor and display roll number, name, and marks using `DBMS_OUTPUT.PUT_LINE`.

2. Exception Handling Program:

1. Declare variables for division.
2. Perform division inside a `BEGIN` block.
3. Handle errors using `EXCEPTION` block (`ZERO_DIVIDE` and `OTHERS`).
4. Display appropriate messages for errors.

Program:

Program 1: Cursor Example

```
-- Enable server output
SET SERVEROUTPUT ON;


-- Create sample table
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE students';
EXCEPTION
   WHEN OTHERS THEN

      NULL;  -- Ignore error if table does not exist
END;
/

CREATE TABLE students (

   roll_no NUMBER PRIMARY KEY,
   name    VARCHAR2(50),
   marks   NUMBER

);
```

```
-- Insert sample data

INSERT INTO students VALUES (101, 'Rahul', 85);
INSERT INTO students VALUES (102, 'Priya', 72);
INSERT INTO students VALUES (103, 'Amit', 59);
INSERT INTO students VALUES (104, 'Sneha', 40);


COMMIT;

-- Cursor Program
DECLARE
   CURSOR stu_cur IS

      SELECT roll_no, name, marks
      FROM students;


   v_roll   students.roll_no%TYPE;
   v_name   students.name%TYPE;
   v_marks  students.marks%TYPE;
BEGIN

   DBMS_OUTPUT.PUT_LINE('--- Student Details Using Cursor ---');
   OPEN stu_cur;

   LOOP

      FETCH stu_cur INTO v_roll, v_name, v_marks;
      EXIT WHEN stu_cur%NOTFOUND;


      DBMS_OUTPUT.PUT_LINE('Roll: ' || v_roll ||
                    ' | Name: ' || v_name ||
                    ' | Marks: ' || v_marks);

   END LOOP;

   CLOSE stu_cur;

END;

/
```

Program 2: Exception Handling Example

```
-- Enable server output
SET SERVEROUTPUT ON;
```

```
DECLARE

   num1   NUMBER := 10;

   num2   NUMBER := 0;   -- This will cause division by zero
   result NUMBER;
BEGIN

   DBMS_OUTPUT.PUT_LINE('--- Exception Handling Example ---');


   -- Attempt division
   result := num1 / num2;
   DBMS_OUTPUT.PUT_LINE('Result: ' || result);


EXCEPTION

   WHEN ZERO_DIVIDE THEN

      DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed!');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Some other error occurred: ' || SQLERRM);
END;
/
```

<u>Output:</u>

1. Cursor Output

```
--- Student Details Using Cursor ---
Roll: 101 | Name: Rahul | Marks: 85
Roll: 102 | Name: Priya | Marks: 72
Roll: 103 | Name: Amit | Marks: 59
Roll: 104 | Name: Sneha | Marks: 40


PL/SQL procedure successfully completed.
```

2. Exception Output

```
--- Exception Handling Example ---
Error: Division by zero is not allowed!


PL/SQL procedure successfully completed.
```

<u>Result:</u> Both programs executed successfully, demonstrating cursors and exception handling in PL/SQL.

Experiment – 10

Implement Integrity Constraints

Aim: To implement integrity constraints in SQL to ensure data accuracy and consistency.

Algorithm/ Procedure:

1. Create a database and tables (`Students` and `Marks`).
2. Apply **Primary Key, Foreign Key, Not Null, Unique, and Check** constraints while creating tables.
3. Insert sample data into the tables.
4. Verify that constraints prevent invalid data insertion.
5. Retrieve data using `SELECT` to confirm successful insertion.

Program:

```
-- Create Database (optional)

CREATE DATABASE SchoolDB;
USE SchoolDB;



-- Create Table with Integrity Constraints

CREATE TABLE Students (

   StudentID   INT PRIMARY KEY,            -- Primary Key Constraint
   Name        VARCHAR(50) NOT NULL,       -- NOT NULL Constraint
   Age         INT CHECK (Age >= 5 AND Age <= 25),  -- CHECK Constraint
   Email       VARCHAR(100) UNIQUE         -- UNIQUE Constraint
);

CREATE TABLE Marks (

   MarkID      INT PRIMARY KEY,            -- Primary Key Constraint
   StudentID   INT,                        -- Foreign Key Constraint
   Subject     VARCHAR(50) NOT NULL,
   Marks       INT CHECK (Marks >= 0 AND Marks <= 100),

   CONSTRAINT FK_Student FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);
```

-- Insert Sample Data

```
INSERT INTO Students VALUES (1, 'Rahul', 15, 'rahul@example.com');
INSERT INTO Students VALUES (2, 'Priya', 16, 'priya@example.com');
INSERT INTO Students VALUES (3, 'Amit', 14, 'amit@example.com');
INSERT INTO Marks VALUES (101, 1, 'Math', 85);

INSERT INTO Marks VALUES (102, 2, 'Science', 90);

INSERT INTO Marks VALUES (103, 3, 'English', 78);
```

-- Verify Data

```
SELECT * FROM Students;
SELECT * FROM Marks;
```

Output:

1. Student Table

StudentID	Name	Age	Email
1	Rahul	15	rahul@example.com
2	Priya	16	priya@example.com
3	Amit	14	amit@example.com
NULL	NULL	NULL	NULL

2. Marks Table

MarkID	StudentID	Subject	Marks
101	1	Math	85
102	2	Science	90
103	3	English	78
NULL	NULL	NULL	NULL

Result: Tables were created successfully with all integrity constraints; valid data was inserted; constraints ensured data accuracy, uniqueness, and referential integrity; output displayed correctly.

Experiment – 11

Implement First, Second and Third normalization techniques

Aim: To implement First, Second, and Third Normal Forms (1NF, 2NF, 3NF) to organize a database and remove redundancy, partial, and transitive dependencies.

Algorithm/ Procedure:

1. Create an **unnormalized table (UNF)** with repeating groups and redundant data.
2. Apply **1NF**: Remove repeating groups and make all columns atomic.
3. Apply **2NF**: Eliminate partial dependency by creating separate tables for entities (Student, Course, StudentCourse).
4. Apply **3NF**: Eliminate transitive dependency by separating dependent attributes (Instructor table).
5. Insert sample data at each stage and verify using `SELECT`.

Program:

Step 1: Create Unnormalized Table (UNF)

```
-- Enable server output if using PL/SQL environment
SET SERVEROUTPUT ON;
-- Unnormalized table (UNF)
CREATE TABLE StudentCourseUNF (
   StudentID   INT,
   StudentName VARCHAR(50),
   CourseIDs   VARCHAR(50),  -- multiple courses in one column (comma-separated)
   CourseNames VARCHAR(100), -- multiple course names in one column
   Instructor  VARCHAR(50)

);


-- Insert sample data

INSERT INTO StudentCourseUNF VALUES (1, 'Rahul', 'C1,C2', 'Math,Science', 'Mr. Sharma');
INSERT INTO StudentCourseUNF VALUES (2, 'Priya', 'C1,C3', 'Math,English', 'Mr. Sharma');
INSERT INTO StudentCourseUNF VALUES (3, 'Amit', 'C2', 'Science', 'Mrs. Verma');


SELECT * FROM StudentCourseUNF;
```

Step 2: First Normal Form (1NF)

```
-- 1NF: Separate each course into a new row
CREATE TABLE StudentCourse1NF (
   StudentID   INT,
```

```
   StudentName VARCHAR(50),
   CourseID    VARCHAR(10),
   CourseName  VARCHAR(50),
   Instructor  VARCHAR(50)
);

-- Insert atomic data

INSERT INTO StudentCourse1NF VALUES (1, 'Rahul', 'C1', 'Math', 'Mr. Sharma');
INSERT INTO StudentCourse1NF VALUES (1, 'Rahul', 'C2', 'Science', 'Mr. Sharma');
INSERT INTO StudentCourse1NF VALUES (2, 'Priya', 'C1', 'Math', 'Mr. Sharma');
INSERT INTO StudentCourse1NF VALUES (2, 'Priya', 'C3', 'English', 'Mr. Sharma');
INSERT INTO StudentCourse1NF VALUES (3, 'Amit', 'C2', 'Science', 'Mrs. Verma');

SELECT * FROM StudentCourse1NF;
```

Step 3: Second Normal Form (2NF)

```
-- Student Table
CREATE TABLE Student (
   StudentID   INT PRIMARY KEY,
   StudentName VARCHAR(50)

);

INSERT INTO Student VALUES (1, 'Rahul');
INSERT INTO Student VALUES (2, 'Priya');
INSERT INTO Student VALUES (3, 'Amit');
-- Course Table
CREATE TABLE Course (
   CourseID   VARCHAR(10) PRIMARY KEY,

   CourseName VARCHAR(50),
   Instructor VARCHAR(50)
);

INSERT INTO Course VALUES ('C1', 'Math', 'Mr. Sharma');

INSERT INTO Course VALUES ('C2', 'Science', 'Mrs. Verma');
INSERT INTO Course VALUES ('C3', 'English', 'Mr. Sharma');

-- StudentCourse Table

CREATE TABLE StudentCourse2NF (
   StudentID INT,
```

```
    CourseID  VARCHAR(10),

    PRIMARY KEY (StudentID, CourseID),

    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
INSERT INTO StudentCourse2NF VALUES (1, 'C1');
INSERT INTO StudentCourse2NF VALUES (1, 'C2');
INSERT INTO StudentCourse2NF VALUES (2, 'C1');
INSERT INTO StudentCourse2NF VALUES (2, 'C3');
INSERT INTO StudentCourse2NF VALUES (3, 'C2');
SELECT * FROM Student;
SELECT * FROM Course;
SELECT * FROM StudentCourse2NF;
```

Step 4: Third Normal Form (3NF)

```
-- Instructor Table
CREATE TABLE Instructor (
    InstructorID INT PRIMARY KEY,
    InstructorName VARCHAR(50)
);


INSERT INTO Instructor VALUES (1, 'Mr. Sharma');
INSERT INTO Instructor VALUES (2, 'Mrs. Verma');

-- Updated Course Table with InstructorID
CREATE TABLE Course3NF (
    CourseID   VARCHAR(10) PRIMARY KEY,

    CourseName VARCHAR(50),
    InstructorID INT,
    FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID)

);


INSERT INTO Course3NF VALUES ('C1', 'Math', 1);
INSERT INTO Course3NF VALUES ('C2', 'Science', 2);
INSERT INTO Course3NF VALUES ('C3', 'English', 1);
SELECT * FROM Student;
SELECT * FROM Instructor;
SELECT * FROM Course3NF;
SELECT * FROM StudentCourse2NF;
```

Output:

Step 1: Create Unnormalized Table (UNF)

	STUDENTID	STUDENTNAME	COURSEIDS	COURSENAMES
1	1	Rahul	C1,C2	Math,Science
2	2	Priya	C1,C3	Math,English
3	3	Amit	C2	Science

Step 2: First Normal Form (1NF)

	STUDENTID	STUDENTNAME	COURSEID	COURSENAME
1	1	Rahul	C1	Math
2	1	Rahul	C2	Science
3	2	Priya	C1	Math
4	2	Priya	C3	English

Step 3: Second Normal Form (2NF)

	STUDENTID	STUDENTNAME
1	1	Rahul
2	2	Priya
3	3	Amit

Step 4: Third Normal Form (3NF)

	STUDENTID	STUDENTNAME
1	1	Rahul
2	2	Priya
3	3	Amit

Result: **Tables were successfully normalized into** 1NF, 2NF, and 3NF**; redundancy and anomalies were reduced; data is organized with atomic, consistent, and dependent attributes correctly separated; all queries executed successfully.**

Experiment – 12

Implement Fourth and Fifth form of normalization techniques

Aim: To implement Fourth and Fifth Normal Forms (4NF & 5NF) to eliminate multi-valued and join dependencies, ensuring data consistency and reducing redundancy.

Algorithm/ Procedure:

1. 4NF

1. Create an unnormalized table with multi-valued attributes (e.g., Skills and Hobbies for students).
2. Separate the independent multi-valued attributes into different tables (`StudentSkills`, `StudentHobbies`).
3. Insert sample data and verify using `SELECT`.

2. 5NF

4. Identify join dependencies (e.g., Student, Course, Instructor).
5. Create separate tables for entities and a junction table (`StudentCourseInstructor`) to handle many-to-many relationships.
6. Insert sample data and verify by reconstructing relationships via joins.

Program:

Step 1: Create Sample Table with Multi-Valued Dependencies

```
-- Enable server output
SET SERVEROUTPUT ON;
-- 4NF: A table where a student can have multiple skills and multiple hobbies (multi-valued
dependency)

CREATE TABLE StudentAttributesUNF (
   StudentID   INT,
   StudentName VARCHAR(50),
   Skill       VARCHAR(50),
   Hobby       VARCHAR(50)
);
-- Insert sample data (repeating combinations)

INSERT INTO StudentAttributesUNF VALUES (1, 'Rahul', 'C', 'Chess');
INSERT INTO StudentAttributesUNF VALUES (1, 'Rahul', 'C', 'Football');
INSERT INTO StudentAttributesUNF VALUES (1, 'Rahul', 'Java', 'Chess');
INSERT INTO StudentAttributesUNF VALUES (1, 'Rahul', 'Java', 'Football');
INSERT INTO StudentAttributesUNF VALUES (2, 'Priya', 'Python', 'Reading');
INSERT INTO StudentAttributesUNF VALUES (2, 'Priya', 'Python', 'Music');

SELECT * FROM StudentAttributesUNF;
```

Step 2: Fourth Normal Form (4NF)

```sql
-- Student Table

CREATE TABLE Student4NF (
   StudentID   INT PRIMARY KEY,
   StudentName VARCHAR(50));


INSERT INTO Student4NF VALUES (1, 'Rahul');
INSERT INTO Student4NF VALUES (2, 'Priya');


-- StudentSkills Table
CREATE TABLE StudentSkills (
   StudentID INT,

   Skill     VARCHAR(50),

   PRIMARY KEY (StudentID, Skill),
   FOREIGN KEY (StudentID) REFERENCES Student4NF(StudentID)

);


INSERT INTO StudentSkills VALUES (1, 'C');

INSERT INTO StudentSkills VALUES (1, 'Java');
INSERT INTO StudentSkills VALUES (2, 'Python');


-- StudentHobbies Table
CREATE TABLE StudentHobbies (
   StudentID INT,

   Hobby     VARCHAR(50),

   PRIMARY KEY (StudentID, Hobby),

   FOREIGN KEY (StudentID) REFERENCES Student4NF(StudentID)

);


INSERT INTO StudentHobbies VALUES (1, 'Chess');
INSERT INTO StudentHobbies VALUES (1, 'Football');
INSERT INTO StudentHobbies VALUES (2, 'Reading');
INSERT INTO StudentHobbies VALUES (2, 'Music');
SELECT * FROM Student4NF;
SELECT * FROM StudentSkills;
SELECT * FROM StudentHobbies;
```

Step 3: Fifth Normal Form (5NF)

```
-- Suppose a student can enroll in multiple courses and each course has multiple instructors
CREATE TABLE Student5NF (
   StudentID INT PRIMARY KEY,

   StudentName VARCHAR(50)

);


CREATE TABLE Course5NF (

   CourseID VARCHAR(10) PRIMARY KEY,

   CourseName VARCHAR(50)

);


CREATE TABLE Instructor5NF (
   InstructorID INT PRIMARY KEY,
   InstructorName VARCHAR(50)
);


-- Many-to-many relationships handled via junction table
CREATE TABLE StudentCourseInstructor (
   StudentID    INT,

   CourseID     VARCHAR(10),

   InstructorID INT,

   PRIMARY KEY (StudentID, CourseID, InstructorID),

   FOREIGN KEY (StudentID) REFERENCES Student5NF(StudentID),
   FOREIGN KEY (CourseID) REFERENCES Course5NF(CourseID),
   FOREIGN KEY (InstructorID) REFERENCES Instructor5NF(InstructorID)
);

-- Insert sample data

INSERT INTO Student5NF VALUES (1, 'Rahul');
INSERT INTO Course5NF VALUES ('C1', 'Math');
INSERT INTO Instructor5NF VALUES (1, 'Mr. Sharma');
INSERT INTO StudentCourseInstructor VALUES (1, 'C1', 1);
SELECT * FROM Student5NF;
SELECT * FROM Course5NF;
SELECT * FROM Instructor5NF;
```

SELECT * FROM StudentCourseInstructor;

Output:

Step 1: Create Sample Table with Multi-Valued Dependencies

	STUDENTID	STUDENTNAME	SKILL	HOBBY
1	1	Rahul	C	Chess
2	1	Rahul	C	Football
3	1	Rahul	Java	Chess
4	1	Rahul	Java	Football

Step 2: Fourth Normal Form (4NF)

	STUDENTID	STUDENTNAME
1	1	Rahul
2	2	Priya

Step 3: Fifth Normal Form (5NF)

	STUDENTID	STUDENTNAME
1	1	Rahul

<u>Result:</u> All queries executed successfully and returned consistent, accurate results

Experiment – 13

Implement the functions/procedures to begin, commit, and rollback transactions.

Aim: To implement transactions in PL/SQL using `BEGIN`, `COMMIT`, and `ROLLBACK` to manage data consistency during database operations.

Algorithm/ Procedure:

1. Create an **Account** table with sample data (`AccountID`, `AccountName`, `Balance`).
2. Create a **PL/SQL procedure** `TransferAmount` to transfer funds between accounts.
3. Inside the procedure, check the sender's balance.
4. If the balance is sufficient, **deduct and add the amount**, then `COMMIT` the transaction.
5. If the balance is insufficient or an error occurs, **ROLLBACK** the transaction.
6. Execute the procedure with different test cases and verify the balances using `SELECT`.

Program:

Step 1: Create Sample Table

```
-- Enable server output
SET SERVEROUTPUT ON;


-- Drop table if exists
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE Account';
EXCEPTION
   WHEN OTHERS THEN
      NULL;
END;

/


-- Create Account Table
CREATE TABLE Account (
   AccountID   INT PRIMARY KEY,

   AccountName VARCHAR(50),
   Balance     NUMBER
);


-- Insert sample data

INSERT INTO Account VALUES (1, 'Rahul', 5000);
INSERT INTO Account VALUES (2, 'Priya', 3000);
COMMIT;
```

```sql
SELECT * FROM Account;
```

Step 2: Create Procedures for Transactions

```sql
-- Procedure to transfer amount (with transaction control)
CREATE OR REPLACE PROCEDURE TransferAmount(
   p_FromAccount INT,
   p_ToAccount   INT,
   p_Amount      NUMBER
)
IS
   v_FromBalance NUMBER;
BEGIN
   -- BEGIN Transaction implicitly in PL/SQL block

   -- Get balance of from account

   SELECT Balance INTO v_FromBalance FROM Account WHERE AccountID =
p_FromAccount;
   IF v_FromBalance < p_Amount THEN

      DBMS_OUTPUT.PUT_LINE('Insufficient balance! Transaction will be rolled back.');
      ROLLBACK;  -- Rollback transaction
   ELSE

      -- Deduct from sender
      UPDATE Account
      SET Balance = Balance - p_Amount
      WHERE AccountID = p_FromAccount;


      -- Add to receiver
      UPDATE Account
      SET Balance = Balance + p_Amount
      WHERE AccountID = p_ToAccount;


      DBMS_OUTPUT.PUT_LINE('Transaction successful! Committing changes...');
      COMMIT;  -- Commit transaction
   END IF;




EXCEPTION

   WHEN OTHERS THEN

      DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
```

```
    ROLLBACK;  -- Rollback on any error
END;
/
```

Step 3: Execute the Procedure

```
-- Successful transaction

EXEC TransferAmount(1, 2, 2000);


-- Transaction with insufficient balance (will rollback)
EXEC TransferAmount(2, 1, 5000);


-- Verify final balances
SELECT * FROM Account;
```

Output:

Step 1: Create Sample Table

	ACCOUNTID	ACCOUNTNAME	BALANCE
1	1	Rahul	5000
2	2	Priya	3000

Step 2: Create Procedures for Transactions

```
Procedure TRANSFERAMOUNT compiled
```

Step 3: Execute the Procedure

Download Execution time: 0.004 seconds

	ACCOUNTID	ACCOUNTNAME	BALANCE
1	1	Rahul	8000
2	2	Priya	0

Result: Transactions executed successfully; amounts were transferred correctly when balance was sufficient.

Experiment – 14

Analyze the structure and properties of B-tree index and its variants

Aim: To analyze the structure and properties of B-tree indexes and their variants (normal, unique, composite, and function-based) in MySQL.

Algorithm/ Procedure:

1. Created an `Employee` table and inserted sample data.
2. Applied different types of B-tree indexes:
3. Used `SHOW INDEXES`, `information_schema`, and `EXPLAIN` queries to study index structure and performance.
4. Executed queries with filtering conditions to observe how the optimizer uses indexes.

Program:

```
-- 1. Create Employee
table CREATE TABLE
Employee (
   EmpID INT PRIMARY KEY AUTO_INCREMENT,
   EmpName VARCHAR(50),
   Department
   VARCHAR(50), Salary
   INT
);


-- 2. Insert sample data

INSERT INTO Employee (EmpName, Department, Salary)
VALUES ('Alice', 'HR', 60000),
('Bob', 'Finance', 55000),

('Charlie', 'HR', 70000),

('David', 'IT', 75000),

('Eva', 'Finance', 65000),
('Frank', 'IT', 72000),

('Grace', 'HR', 50000);


-- 3. Normal B-tree index on Salary

CREATE INDEX idx_salary ON Employee(Salary);
```

-- 4. Unique B-tree index on EmpName

CREATE UNIQUE INDEX idx_empname ON Employee(EmpName);

-- 5. Composite (multi-column) B-tree index on Department and Salary
CREATE INDEX idx_dept_salary ON Employee(Department, Salary);

-- 6. Function-based index (MySQL uses generated column
instead) ALTER TABLE Employee
ADD COLUMN EmpNameUpper VARCHAR(50) GENERATED ALWAYS AS (UPPER(EmpName))
STORED,

ADD INDEX idx_upper_empname (EmpNameUpper);

-- 7. Show table structure with indexes
SHOW CREATE TABLE Employee;

-- 8. Show all indexes on the Employee
table SHOW INDEXES FROM Employee;

-- 9. Check index type (BTREE, HASH, etc.)

SELECT INDEX_NAME, COLUMN_NAME, INDEX_TYPE

FROM information_schema.STATISTICS
WHERE TABLE_NAME = 'Employee';

-- 10. Analyze table to update index statistics
ANALYZE TABLE Employee;

-- 11. Use EXPLAIN to see how indexes are used in
queries EXPLAIN SELECT * FROM Employee WHERE
Salary > 60000;
EXPLAIN SELECT * FROM Employee WHERE Department = 'HR' AND Salary >
60000; EXPLAIN SELECT * FROM Employee WHERE EmpNameUpper = 'ALICE';

<u>Output:</u>

1. Result Table – 1.1

Table	Create Table
Employee	CREATE TABLE `employee` (`EmpID` int NO...

1.2

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_
employee	0	PRIMARY	1	EmpID	A	0	NULL	NULL		BTREE		
employee	0	idx_empname	1	EmpName	A	0	NULL	NULL	YES	BTREE		
employee	1	idx_salary	1	Salary	A	0	NULL	NULL	YES	BTREE		
employee	1	idx_dept_salary	1	Department	A	0	NULL	NULL	YES	BTREE		
employee	1	idx_dept_salary	2	Salary	A	0	NULL	NULL	YES	BTREE		
employee	1	idx_upper_empname	1	EmpNameUpper	A	0	NULL	NULL	YES	BTREE		

1.3

Table	Op	Msg_type	Msg_text
schooldb.employee	analyze	status	OK

1.4

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Employee	NULL	range	idx_salary	idx_salary	5	NULL	4	100.00	Using index condition

1.5

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Employee	NULL	range	idx_salary,idx_dept_salary	idx_dept_salary	208	NULL	1	100.00	Using index condition

1.6

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Employee	NULL	ref	idx_upper_empname	idx_upper_empname	203	const	1	100.00	NULL

2. Statistics Table

INDEX_NAME	COLUMN_NAME	INDEX_TYPE
dept_id	dept_id	BTREE
phone_number	phone_number	BTREE
PRIMARY	emp_id	BTREE
idx_dept_salary	Department	BTREE
idx_dept_salary	Salary	BTREE
idx_empname	EmpName	BTREE
idx_salary	Salary	BTREE

Result B-tree indexes improved query performance, ensured uniqueness, optimized multi-column searches, supported function-based lookups, and all were confirmed as BTREE type in MySQL.

Experiment – 15

Case Study: Analyze different types of failures such as transaction failures, system crashes, and disk failures

Aim: To analyze different types of failures in a database system, including transaction failures, system crashes, and disk failures, and observe how MySQL ensures data consistency and recovery.

Algorithm/ Procedure:

1. Created a sample `Accounts` table with sample data.
2. Simulated a **transaction failure** by starting a transaction, performing updates, and introducing an error to trigger rollback.
3. Explained **system crash recovery**, where InnoDB automatically recovers committed transactions using redo logs.
4. Discussed **disk failure handling** using backups and binary logs for data restoration.
5. Verified table state after transaction failure using `SELECT` queries.

Program:

1. Create Table and Insert Data

```
CREATE TABLE Accounts (
   AccountID INT PRIMARY KEY,
   AccountName VARCHAR(50),
   Balance DECIMAL(10,2)
);


INSERT INTO Accounts (AccountID, AccountName, Balance) VALUES
(1, 'Alice', 1000.00),
(2, 'Bob', 1500.00);
```

2. Simulate Transaction Failure

```
START TRANSACTION;


UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;


-- Simulate error

-- SET @x = 1/0;  -- Uncomment to simulate failure


COMMIT;
```

3. System Crash / Recovery

SELECT * FROM Accounts;

4. Disk Failure Handling

SHOW BINARY LOGS;

Output:

1. Create Table and Insert Data

Result Grid | Filter Rows: | Edit:

AccountID	AccountName	Balance
1	Alice	1000.00
2	Bob	1500.00
NULL	NULL	NULL

2. Simulate Transaction Failure

AccountID	AccountName	Balance
1	Alice	500.00
2	Bob	1500.00
NULL	NULL	NULL

3. System Crash / Recovery

Result Grid | Filter Rows: | Edit:

AccountID	AccountName	Balance
1	Alice	1000.00
2	Bob	1500.00
NULL	NULL	NULL

4. Disk Failure Handling

Log_name	File_size	Encrypted
DESKTOP-3GK020J-bin.000914	157	No
DESKTOP-3GK020J-bin.000915	157	No
DESKTOP-3GK020J-bin.000916	157	No
DESKTOP-3GK020J-bin.000917	157	No
DESKTOP-3GK020J-bin.000918	157	No
DESKTOP-3GK020J-bin.000919	157	No
DESKTOP-3GK020J-bin.000920	157	No

Result: Transaction failures caused automatic rollback, system crashes were recovered by InnoDB, and disk failures can be restored using backups and logs, ensuring data consistency.