DELHI TECHNICAL CAMPUS
GREATER NOIDA
Affiliated to GGSIPU and Approved by AICTE & COA
Department of Computer Science & Engineering

EXPERIMENT - 01

Q. Experiments based on DDL commands - CREATE, ALTER, DROP and TRUNCATE.

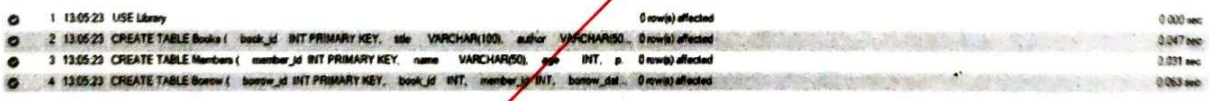
</> INPUT QUERIES </>

1. CREATE

```
CREATE TABLE Books (
  book_id   INT PRIMARY KEY,
  title     VARCHAR(100),
  author    VARCHAR(50),
  published_year INT,
  genre     VARCHAR(30)
);

CREATE TABLE Members (
  member_id  INT PRIMARY KEY,
  name       VARCHAR(50),
  age        INT,
  phone      VARCHAR(10)
);

CREATE TABLE Borrow (
  borrow_id  INT PRIMARY KEY,
  book_id    INT,
  member_id  INT,
  borrow_date DATE,
  return_date DATE,
  FOREIGN KEY (book_id) REFERENCES Books(book_id),
  FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
```



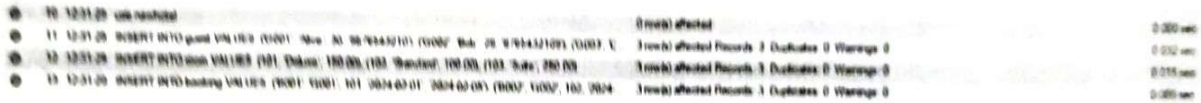
2. INSERT Sample Data

```
INSERT INTO guest VALUES
('G001', 'Alice', 30, '9876543210'),
('G002', 'Bob', 28, '8765432109'),
('G003', 'Charlie', 35, '7654321098');

INSERT INTO room VALUES
(101, 'Deluxe', 150.00),
```

```
(102, 'Standard', 100.00),
(103, 'Suite', 250.00);

INSERT INTO booking VALUES
('B001', 'G001', 101, '2024-02-01', '2024-02-05'),
('B002', 'G002', 102, '2024-02-02', '2024-02-06'),
('B003', 'G003', 103, '2024-02-03', '2024-02-07');
```



3. Queries for Joins and Subqueries

(i). Join Query (Retrieve guest details along with their booking and room type)

```
SELECT g.guest_id, g.name, g.age, g.phone, b.booking_id, b.room_no, r.type, r.price
FROM guest g
JOIN booking b ON g.guest_id = b.guest_id
JOIN room r ON b.room_no = r.room_no;
```

	guest_id	name	age	phone	booking_id	room_no	type	price
▶	G001	Alice	30	9876543210	B001	101	Deluxe	150.00
	G002	Bob	28	8765432109	B002	102	Standard	100.00
	G003	Charlie	35	7654321098	B003	103	Suite	250.00

(ii). Subquery (Find guests who booked the most expensive room)

```
SELECT name FROM guest
WHERE guest_id IN (
  SELECT guest_id FROM booking
  WHERE room_no = (
    SELECT room_no FROM room ORDER BY price DESC LIMIT 1
  )
);
```

	name
▶	Charlie

(iii). Subquery with Join (Find guests who booked a Deluxe room)

```
SELECT name FROM guest
WHERE guest_id IN (
  SELECT guest_id FROM booking
  WHERE room_no IN (SELECT room_no FROM room WHERE type = 'Deluxe')
);
```

	name
▶	Alice

(iv). Join with Aggregation (Count of bookings per guest)

```
SELECT g.name, COUNT(b.booking_id) AS total_bookings
FROM guest g
LEFT JOIN booking b ON g.guest_id = b.guest_id
GROUP BY g.name;
```

	name	total_bookings
▶	Alice	1
	Bob	1
	Charlie	1

EXPERIMENT - 2

Aim: Apply the integrity constraints like Primary Key, Foreign key, Check, NOT NULL, etc. to the tables.

Integrity constraints:
Integrity constraints are rules enforced on database tables to maintain accuracy and consistency of data.

1. Primary Key (PK)

- Ensures each row in a table has a unique and non-null identifier.
- A table can have only one primary key, which may consist of one or more columns (composite key).

Example:

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(100),
   Age INT
);
```

2. Foreign Key (FK)

- Ensures a relationship between two tables by enforcing referential integrity.
- The foreign key in one table must reference a primary key in another table.

Example:

```
CREATE TABLE Enrollments (
   EnrollmentID INT PRIMARY KEY,
   StudentID INT,
   CourseID INT,
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
   FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

3. NOT NULL

- Ensures that a column cannot have NULL values.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Salary DECIMAL(10,2) NOT NULL
);
```

4. CHECK Constraint

- Ensures that values in a column meet specific conditions.

Example:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    Price DECIMAL(10,2) CHECK (Price > 0),
    Stock INT CHECK (Stock >= 0)
);
```

5. Unique Constraint

- Ensures that all values in a column (or group of columns) are distinct.

Example:

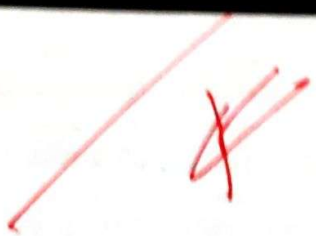
```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Email VARCHAR(255) UNIQUE
);
```

PROGRAM:

```
postgres=# CREATE TABLE users (
postgres(# user_id SERIAL PRIMARY KEY,
postgres(# username VARCHAR(50) UNIQUE NOT NULL,
postgres(# age INT CHECK (age>=18));
CREATE TABLE
postgres=# INSERT INTO users
postgres=# VALUES (1, 'David', 27),
postgres=# (2, 'Josh', 28),
postgres=# (3, 'Radha', 30),
postgres=# (4, 'Jessica', 25);
INSERT 0 4
postgres=# SELECT * FROM users;
 user_id | username | age
---------+----------+-----
       1 | David    |  27
       2 | Josh     |  28
       3 | Radha    |  30
       4 | Jessica  |  25
(4 rows)
```

```
postgres=# CREATE TABLE orders (
postgres(# order_id SERIAL PRIMARY KEY,
postgres(# user_id INT NOT NULL,
postgres(# order_item VARCHAR(50),
postgres(# FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE);
CREATE TABLE
postgres=# INSERT INTO orders
postgres-# VALUES (101, 1 , 'burger'),
postgres-# (102, 2, 'burger'),
postgres-# (103, 3, 'coffee'),
postgres-# (104, 4, 'pizza');
INSERT 0 4
postgres=# SELECT * FROM orders;
 order_id | user_id | order_item
----------+---------+------------
      101 |       1 | burger
      102 |       2 | burger
      103 |       3 | coffee
      104 |       4 | pizza
(4 rows)
```

```
postgres=# \d orders
                                  Table "public.orders"
   Column   |          Type          | Collation | Nullable |                Default
------------+------------------------+-----------+----------+---------------------------------------
 order_id   | integer                |           | not null | nextval('orders_order_id_seq'::regclass)
 user_id    | integer                |           | not null |
 order_item | character varying(50)  |           |          |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id)
Foreign-key constraints:
    "orders_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
```



EXPERIMENT - 03

Q. Experiment based on basic DML commands - SELECT, INSERT, UPDATE and DELETE.

</> INPUT QUERIES </>

	guest_id	guest_name	age	city	check_in_date	room_type	amount_paid
▶	1	John Doe	30	New York	2025-01-01	Single	200.00
	2	Jane Smith	25	Los Angeles	2025-02-01	Double	150.00
	3	Jim Brown	35	Chicago	2025-02-05	Single	180.00
	4	Jack White	40	New York	2025-01-10	Suite	300.00
	5	Jill Green	28	Chicago	2025-02-10	Double	170.00
	6	Jerry Black	33	Los Angeles	2025-01-15	Single	220.00
	7	Joyce Pink	22	New York	2025-02-01	Suite	350.00
	8	Jordan Blue	45	Chicago	2025-01-30	Single	190.00
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

1. SELECT - Retrieve Data from the Table

```
– Retrieve all columns and rows from the guest table
Use mydb;
SELECT * FROM guest;

– Retrieve specific columns (e.g., guest_name, city, and room_type)
SELECT guest_name, city, room_type FROM guest;

– Retrieve guests who paid more than $200
SELECT guest_name, amount_paid
FROM guest
WHERE amount_paid > 200;

– Retrieve the number of guests in New York
SELECT COUNT(*) AS num_guests_in_ny
FROM guest
WHERE city = 'New York';
```

	num_guests_in_ny
▶	3

2. INSERT - Add New Data into the Table

```
USE MYDB;
– Insert a new guest record into the guest table
INSERT INTO guest (guest_id, guest_name, age, city, check_in_date, room_type, amount_paid)
VALUES (9, 'Alice Brown', 26, 'Los Angeles', '2025-02-15', 'Double', 180.00);
```

24 12:58:10 INSERT INTO guest (guest_id, guest_name, age, city, check_in_date, room_type, amount_... 1 row(s) affected 0.015 sec

3. **UPDATE** - Modify Existing Data

```
-- Update the amount paid by guest with guest_id 3
UPDATE guest
SET amount_paid = 200.00
WHERE guest_id = 3;

-- Update the room type for guests in New York who are under 30 years old
UPDATE guest
SET room_type = 'Penthouse'
WHERE city = 'New York' AND age < 30;
```

4. **DELETE** - Remove Data from the Table

```
-- Delete the record of the guest with guest_id 9
DELETE FROM guest
WHERE guest_id = 9;

-- Delete all guests from Chicago who paid less than $180
DELETE FROM guest
WHERE city = 'Chicago' AND amount_paid < 180;
```

guest_id	guest_name	age	city	check_in_date	room_type	amount_paid
1	John Doe	30	New York	2025-01-01	Single	200.00
2	Jane Smith	25	Los Angeles	2025-02-01	Double	150.00
3	Jim Brown	35	Chicago	2025-02-05	Single	200.00
4	Jack White	40	New York	2025-01-10	Suite	300.00
5	Jill Green	28	Chicago	2025-02-10	Double	170.00
6	Jerry Black	33	Los Angeles	2025-01-15	Single	220.00
7	Joyce Pink	22	New York	2025-02-01	Suite	350.00
8	Jordan Blue	45	Chicago	2025-01-30	Single	190.00
NULL	NULL	NULL	NULL	NULL	NULL	NULL

EXPERIMENT - 04

Q. Write the queries for implementing BUILT-IN functions, Group By, Having and Order By.

</> INPUT QUERIES </>

1. Table Creation.

```
USE MYDB;

-- Create the guest table
CREATE TABLE guest (
    guest_id INT PRIMARY KEY,
    guest_name VARCHAR(100),
    age INT,
    city VARCHAR(100),
    check_in_date DATE,
    room_type VARCHAR(50),
    amount_paid DECIMAL(10, 2)
);
-- Insert data into the guest table
INSERT INTO guest (guest_id, guest_name, age, city, check_in_date, room_type, amount_paid)
VALUES
(1, 'John Doe', 30, 'New York', '2025-01-01', 'Single', 200.00),
(2, 'Jane Smith', 25, 'Los Angeles', '2025-02-01', 'Double', 150.00),
(3, 'Jim Brown', 35, 'Chicago', '2025-02-05', 'Single', 180.00),
(4, 'Jack White', 40, 'New York', '2025-01-10', 'Suite', 300.00),
(5, 'Jill Green', 28, 'Chicago', '2025-02-10', 'Double', 170.00),
(6, 'Jerry Black', 33, 'Los Angeles', '2025-01-15', 'Single', 220.00),
(7, 'Joyce Pink', 22, 'New York', '2025-02-01', 'Suite', 350.00),
(8, 'Jordan Blue', 45, 'Chicago', '2025-01-30', 'Single', 190.00);
```

guest_id	guest_name	age	city	check_in_date	room_type	amount_paid
1	John Doe	30	New York	2025-01-01	Single	200.00
2	Jane Smith	25	Los Angeles	2025-02-01	Double	150.00
3	Jim Brown	35	Chicago	2025-02-05	Single	180.00
4	Jack White	40	New York	2025-01-10	Suite	300.00
5	Jill Green	28	Chicago	2025-02-10	Double	170.00
6	Jerry Black	33	Los Angeles	2025-01-15	Single	220.00
7	Joyce Pink	22	New York	2025-02-01	Suite	350.00
8	Jordan Blue	45	Chicago	2025-01-30	Single	190.00
NULL	NULL	NULL	NULL	NULL	NULL	NULL

2. SELECT & Group By

```
USE MYDB;
SELECT city, SUM(amount_paid) AS total_amount_paid
FROM guest
```

GROUP BY city;

	city	total_amount_paid
▶	New York	850.00
	Los Angeles	370.00
	Chicago	540.00

3. Having

```
SELECT room_type, SUM(amount_paid) AS total_amount_paid
FROM guest
GROUP BY room_type
HAVING SUM(amount_paid) > 500;
```

	room_type	total_amount_paid
▶	Single	790.00
	Suite	650.00

4. Order By

```
SELECT guest_id, guest_name, amount_paid
FROM guest
ORDER BY amount_paid DESC;
```

	guest_id	guest_name	amount_paid
▶	7	Joyce Pink	350.00
	4	Jack White	300.00
	6	Jerry Black	220.00
	1	John Doe	200.00
	8	Jordan Blue	190.00
	3	Jim Brown	180.00
	5	Jill Green	170.00
	2	Jane Smith	150.00
*	NULL	NULL	NULL

5. Having Condition

```
SELECT city, SUM(amount_paid) AS total_amount_paid
FROM guest
GROUP BY city
HAVING SUM(amount_paid) > 400
ORDER BY total_amount_paid DESC;
```

	city	total_amount_paid
▶	New York	850.00
	Chicago	540.00

Write the queries to implement the Joins and Subqueries.

</> INPUT QUERIES </>

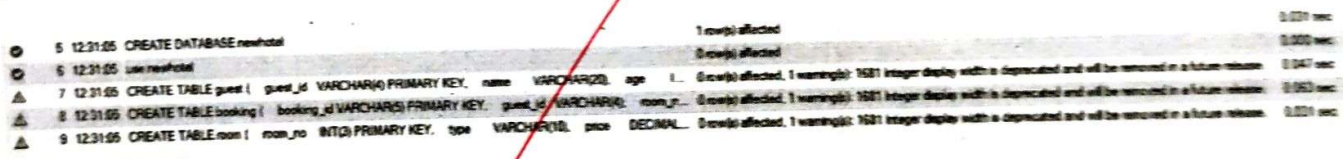
TABLE CREATION

```
USE hotel;

CREATE TABLE guest (
  guest_id  VARCHAR(4) PRIMARY KEY,
  name      VARCHAR(20),
  age       INT(2),
  phone     VARCHAR(10)
);

CREATE TABLE booking (
  booking_id VARCHAR(5) PRIMARY KEY,
  guest_id  VARCHAR(4),
  room_no   INT(3),
  check_in  DATE,
  check_out DATE,
  FOREIGN KEY (guest_id) REFERENCES guest(guest_id)
);

CREATE TABLE room (
  room_no   INT(3) PRIMARY KEY,
  type      VARCHAR(10),
  price     DECIMAL(8,2)
);
```

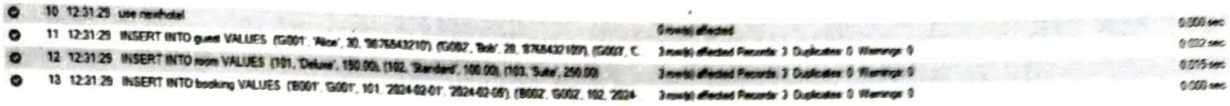


2. INSERT Sample Data

```
INSERT INTO guest VALUES
('G001', 'Alice', 30, '9876543210'),
('G002', 'Bob', 28, '8765432109'),
('G003', 'Charlie', 35, '7654321098');

INSERT INTO room VALUES
(101, 'Deluxe', 150.00),
(102, 'Standard', 100.00),
(103, 'Suite', 250.00);
```

```
INSERT INTO booking VALUES
('B001', 'G001', 101, '2024-02-01', '2024-02-05'),
('B002', 'G002', 102, '2024-02-02', '2024-02-06'),
('B003', 'G003', 103, '2024-02-03', '2024-02-07');
```



3. Queries for Joins and Subqueries

(i). Join Query (Retrieve guest details along with their booking and room type)

```
SELECT g.guest_id, g.name, g.age, g.phone, b.booking_id, b.room_no, r.type, r.price
FROM guest g
JOIN booking b ON g.guest_id = b.guest_id
JOIN room r ON b.room_no = r.room_no;
```

guest_id	name	age	phone	booking_id	room_no	type	price
G001	Alice	30	9876543210	B001	101	Deluxe	150.00
G002	Bob	28	8765432109	B002	102	Standard	100.00
G003	Charlie	35	7654321098	B003	103	Suite	250.00

(ii). Subquery (Find guests who booked the most expensive room)

```
SELECT name FROM guest
WHERE guest_id IN (
  SELECT guest_id FROM booking
  WHERE room_no = (
    SELECT room_no FROM room ORDER BY price DESC LIMIT 1
  )
);
```

name
Charlie

(iii). Subquery with Join (Find guests who booked a Deluxe room)

```
SELECT name FROM guest
WHERE guest_id IN (
  SELECT guest_id FROM booking
  WHERE room_no IN (SELECT room_no FROM room WHERE type = 'Deluxe')
);
```

name
Alice

(iii). Join with Aggregation (Count of bookings per guest)

```
SELECT g.name, COUNT(b.booking_id) AS total_bookings
FROM guest g
LEFT JOIN booking b ON g.guest_id = b.guest_id
GROUP BY g.name;
```

	name	total_bookings
▶	Alice	1
	Bob	1
	Charlie	1

Experiment- 6

AIM : Write the queries to implement the joins and subqueries

Objective: To understand and implement **Joins** and **Subqueries** in SQL.

Theory:

1. Joins in SQL

Joins are used to combine records from two or more tables based on a related column. There are different types of joins in SQL:

INNER JOIN: Returns only matching rows from both tables.

Syntax:

```
SELECT A.ColumnName, B.ColumnName
FROM TableA A
INNER JOIN TableB B
ON A.CommonColumn = B.CommonColumn;
```

LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and matching rows from the right table. If there is no match, NULL values are returned for right table columns.

Syntax:

```
SELECT A.ColumnName, B.ColumnName
FROM TableA A
LEFT JOIN TableB B
ON A.CommonColumn = B.CommonColumn;
```

RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and matching rows from the left table. If there is no match, NULL values are returned for left table columns.

Syntax:

```
SELECT A.ColumnName, B.ColumnName
FROM TableA A
RIGHT JOIN TableB B
ON A.CommonColumn = B.CommonColumn;
```

FULL JOIN (FULL OUTER JOIN): Returns all rows when there is a match in either table. If there is no match, NULL values are returned for columns where there is no match.

Syntax:

```
SELECT A.ColumnName, B.ColumnName
FROM TableA A
FULL JOIN TableB B
ON A.CommonColumn = B.CommonColumn;
```

CROSS JOIN: Returns the Cartesian product of both tables (i.e., each row from the first table is combined with all rows from the second table).

Syntax:

```
SELECT A.ColumnName, B.ColumnName
FROM TableA A
CROSS JOIN TableB B;
```

2. Subqueries in SQL

A **subquery** is a query inside another SQL query. It is used to retrieve data that will be used by the main query. Subqueries can be used with **SELECT, INSERT, UPDATE,** and **DELETE** statements.

- **Subquery in SELECT Statement:**

```
SELECT ColumnName FROM TableName
WHERE ColumnName = (SELECT ColumnName FROM AnotherTable
WHERE Condition);
```

- **Subquery in INSERT Statement:**

```
INSERT INTO TableName (Column1, Column2)
SELECT Column1, Column2 FROM AnotherTable WHERE Condition;
```

- **Subquery in UPDATE Statement:**

```
UPDATE TableName
SET ColumnName = (SELECT ColumnName FROM AnotherTable
WHERE Condition)
WHERE Condition;
```

- **Subquery in DELETE Statement:**

```
DELETE FROM TableName
WHERE ColumnName = (SELECT ColumnName FROM AnotherTable
WHERE Condition);
```

Expected Outcome

By implementing **Joins and Subqueries**, the database ensures:

- Efficient retrieval of related data from multiple tables.
- Enhanced query optimization by using subqueries for filtering and modification.
- Simplified data extraction and relationship establishment between tables.

Conclusion

Joins and Subqueries are powerful SQL concepts that allow effective data retrieval from multiple tables. **Joins** establish relationships between tables based on common keys, while **Subqueries** enable nested queries to refine data selection and manipulation.

Experiment - 7

AIM: Write the queries to implement the set operations and also create views based on views.

Objective:
To understand and implement SQL set operations and view creation using SQL queries.

Theory:
Set operations in SQL are used to combine the results of two or more SELECT statements. These operations must have the same number of columns and compatible data types.

Types of Set Operations:
1. UNION – Combines results and removes duplicates.
2. UNION ALL – Combines results without removing duplicates.
3. INTERSECT – Returns only the common records.
4. MINUS (or EXCEPT) – Returns records from the first SELECT that are not in the second.

Views:
A view is a virtual table that provides a way to simplify complex queries. A view on a view means creating a new view using an existing one.

Implementation:

```
-- Step 1: Create Tables
CREATE TABLE Student (
  ID INT,
  Name VARCHAR(50)
);

CREATE TABLE Alumni (
```

```
  ID INT,
  Name VARCHAR(50)
);

-- Step 2: Insert Sample Data
INSERT INTO Student VALUES (1, 'Aman'), (2, 'Riya'), (3, 'Karan');
INSERT INTO Alumni VALUES (2, 'Riya'), (4, 'Neha');

-- Step 3: Set Operations
SELECT * FROM Student
UNION
SELECT * FROM Alumni;

SELECT * FROM Student
INTERSECT
SELECT * FROM Alumni;

SELECT * FROM Student
EXCEPT
SELECT * FROM Alumni;

-- Step 4: Create Views
CREATE VIEW Student_View AS
SELECT * FROM Student;

CREATE VIEW Top_Student_View AS
SELECT * FROM Student_View WHERE ID < 3;
```

Conclusion:

Set operations help in comparing datasets, while views simplify complex queries. Creating views on views improves data abstraction and reusability.

Experiment - 8

AIM: Demonstrate the concept of Control Structures also demonstrate the concept of Exception Handling.

Objective:

To learn PL/SQL control structures like IF-ELSE, loops, and exception handling blocks.

Theory:

Control Structures in PL/SQL:

1. Conditional Statements: IF...THEN, IF...THEN...ELSE, CASE
2. Loops: FOR loop, WHILE loop, LOOP...EXIT

Exception Handling:

PL/SQL allows trapping runtime errors using EXCEPTION blocks.

Types:

- Predefined exceptions: e.g., ZERO_DIVIDE
- User-defined exceptions

Implementation:

```
DECLARE
  a INT := 10;
  b INT := 0;
  result NUMBER;
BEGIN
  IF a > b THEN
    DBMS_OUTPUT.PUT_LINE('a is greater than b');
  ELSE
    DBMS_OUTPUT.PUT_LINE('b is greater or equal to a');
  END IF;

  FOR i IN 1..5 LOOP
```

```
  DBMS_OUTPUT.PUT_LINE('Loop iteration: ' || i);
 END LOOP;

 BEGIN
  result := a / b;
 EXCEPTION
  WHEN ZERO_DIVIDE THEN
   DBMS_OUTPUT.PUT_LINE('Division by zero occurred');
 END;
END;
```

Conclusion:

Control structures add logic to procedures, and exception handling improves reliability by catching runtime errors.

Experiment - 9

AIM: Create employee management system using MongoDB.

Objective:

To perform MongoDB operations like insert, update, find, and delete in a NoSQL document-based database.

Theory:

MongoDB is a NoSQL database that stores data in JSON-like documents.

Operations in MongoDB:
- insertOne(), find(), updateOne(), deleteOne()
- Uses db.collection syntax.

Implementation:

```
use EmployeeDB;

db.createCollection("employees");

db.employees.insertMany([
  { emp_id: 1, name: "Ravi", dept: "HR", salary: 30000 },
  { emp_id: 2, name: "Sneha", dept: "IT", salary: 50000 },
  { emp_id: 3, name: "Amit", dept: "Sales", salary: 40000 }
]);

db.employees.find();
db.employees.find({ dept: "IT" });

db.employees.updateOne({ emp_id: 2 }, { $set: { salary: 55000 } });

db.employees.deleteOne({ emp_id: 3 });
```

Conclusion:

MongoDB provides a flexible schema to manage employee data and supports CRUD operations efficiently in document-based format.

Experiment - 10

AIM: Connect employee management system using JDBC.

Objective:

To learn how to connect and manipulate data using JDBC with an employee management system.

Theory:

JDBC (Java Database Connectivity) is an API for connecting and executing queries with databases using Java.

Steps:

1. Load JDBC Driver.
2. Establish Connection.
3. Create Statement.
4. Execute Queries.
5. Close Connection.

Implementation:

```
import java.sql.*;
class ConnectDB {
   public static void main(String args[]) {
      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/EmployeeDB", "root",
"password");
         Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
         while (rs.next())
            System.out.println(rs.getInt(1) + " " + rs.getString(2));
```

```
            con.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Conclusion:

JDBC is a standard way to connect Java programs to relational databases and allows data manipulation through Java code.

Experiment - 11

AIM: Executing Queries using NoSQL(document-based)

Objective:
To execute NoSQL queries using document-based operations in MongoDB.

Theory:
NoSQL databases like MongoDB store data as documents. These databases are flexible and ideal for semi-structured or unstructured data.

Common Commands:
- db.collection.find()
- db.collection.insertOne()
- db.collection.updateOne()
- db.collection.deleteOne()

Implementation:

```
use ProductDB;

db.products.insertMany([
  { product_id: 1, name: "Laptop", price: 50000 },
  { product_id: 2, name: "Mouse", price: 500 },
  { product_id: 3, name: "Keyboard", price: 800 }
]);

db.products.find();
db.products.updateOne({ product_id: 2 }, { $set: { price: 600 } });
db.products.deleteOne({ product_id: 3 });
```

Conclusion:

NoSQL document-based queries allow quick data operations with flexible schema, ideal for modern applications.