EXPERIMENT - 6

TITLE :
Write the queries to implement the joins and subqueries.

OBJECTIVES :

- Get a list of all customers who placed orders, along with the order ID and total amount.
- Retrieve all customers, including those who haven't placed any orders.
- List each customer and the total number of orders they have placed using a subquery.
- Find the names of customers who have placed at least one order over $400.
- For each customer, return the order with their highest total amount.

THEORY:

□ JOINs in SQL

□ Definition:
A JOIN is used to combine rows from two or more tables, based on a related column between them.

□ Types of Joins & Their Objectives:

Type of JOIN	Description	Objective
INNER JOIN	Returns only rows with matching values in both tables	Find matching data in two tables (e.g., orders placed by customers)
LEFT JOIN	Returns all rows from the left table, and matched rows from the right table (or NULL if no match)	Include all records from one table, even if they have no match
RIGHT JOIN	Returns all rows from the right table, and matched rows from the left	Like LEFT JOIN, but from the right table's perspective
FULL JOIN	Returns all rows from both tables, matched or not	Show all data even if no match exists
CROSS JOIN	Returns the Cartesian product (all combinations)	Rarely used, but useful for combinations of values

---

□ Objectives of JOINs:

- Combine data spread across multiple tables
- Extract related information (e.g., get customer names with order details)
- Normalize data and avoid redundancy

□ Subqueries in SQL

□ Definition:
A subquery (also known as a nested query) is a query within another SQL query. It's used to compute a value or a set of rows that the outer query depends on.

□ Types of Subqueries & Their Use Cases:

Subquery Location	Use Case	Example Objective

Subquery Location	Use Case	Example Objective
In SELECT clause	To calculate a value for each row	Count how many orders a customer made
In WHERE clause	To filter based on another result	Get customers who made high-value orders
In FROM clause	To treat a subquery as a temporary table	Use aggregated data in a larger query
Correlated Subquery	A subquery that refers to the outer query	Get each customer's most expensive order

---

 Objectives of Subqueries:

Break down complex problems into smaller steps

Perform comparisons or filtering using dynamically generated values

Use as temporary, derived tables

Reuse logic instead of repeating the same code

 JOIN vs Subquery — When to Use What?

Feature	JOIN	Subquery
Combines tables	✓	✗
Filters or calculates	✗	✓
Readability	Clear when combining data	Better for isolating logic
Performance	Usually better with indexed joins	Can be slower if not optimized

---

 Real-life Analogy:

JOIN: Like combining two spreadsheets using a common column (e.g., Customer ID).

Subquery: Like looking up a value in a reference sheet before making a decision (e.g., check if someone ordered more than $500 before showing their name).

EXPERIMENT - 7

TITLE :

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

OBJECTIVES :

Set operations are used to combine results from two or more SELECT statements into a single result set. They are useful when working with similar datasets from different sources/tables and help avoid writing complex joins or loops.

 Theory:
SQL supports four main set operations:

Operation	Description
UNION	Combines results and removes duplicates
UNION ALL	Combines results and keeps duplicates
INTERSECT	Returns only the records common to both SELECT statements
EXCEPT	Returns records from the first SELECT that are not in the second

 Note: Not all databases support INTERSECT and EXCEPT (e.g., MySQL does not, but PostgreSQL and SQL Server do).

 Use Cases / Objectives:

- Merge datasets from multiple regions or categories (e.g., USA vs UK customers)
- Identify overlapping or unique records across tables
- Filter and analyze large datasets split across tables

 Example Objective:
"Find all unique customers across USA and UK tables."
sql
CopyEdit
SELECT Name FROM Customers_USAUNIONSELECT Name FROM Customers_UK;
Objective: Avoid duplicates and merge customers from both countries.

 VIEWS IN SQL
 Objective:
A view is a virtual table that stores the result of a SQL query. It's used to simplify complex queries, improve security, and encapsulate logic for reusability.

 Theory:

A view is not physically stored like a table — it’s just a saved query.

You can use views to:

- ✧ Abstract complicated joins or filters
- ✧ Limit access to certain columns (e.g., for users)
- ✧ Reuse logic in reporting or dashboards

 Creating a Simple View:

sql

CopyEdit

CREATE VIEW Active_Customers ASSELECT * FROM CustomersWHERE IsActive = 1;

Objective:

Hide inactive customers from most queries by creating a reusable view.

 Creating a View Based on Another View (Nested View):

sql

CopyEdit

CREATE VIEW Active_Customers_With_Orders ASSELECT c.Name, o.OrderIDFROM Active_Customers cJOIN Orders o ON c.CustomerID = o.CustomerID;

Objective:

Use the previous view to get a list of only active customers with their orders, reducing code duplication and increasing clarity.

 Benefits / Objectives of Views:

Purpose	Explanation
Simplify Queries	Avoid rewriting complex joins/subqueries
Security & Access	Expose only necessary fields to users
Maintainability	Centralize business logic (like filters) in one place
Reusability	Use views in multiple other queries, reports, and even other views

---

✓Summary

Feature	Objective
Set Operations	Combine or compare query results from multiple SELECT statements
Views	Store complex queries as reusable virtual tables
Nested Views	Build new views on top of existing ones for layered, logical design

EXPERIMENT - 8

TITLE :

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

 Objective:

Control structures are used to control the flow of execution in SQL-based programming languages — like PL/SQL (Oracle), T-SQL (SQL Server), or procedural blocks in PostgreSQL.

These include:

Conditional statements (IF, CASE)

Loops (LOOP, WHILE, FOR)

EXIT / CONTINUE to control loop flow

 Theory:

 IF...THEN...ELSE:

sql

CopyEdit

IF total_sales > 10000 THEN

  bonus := 500;ELSE

  bonus := 200;END IF;

 CASE:

sql

CopyEdit

commission := CASE

```
  WHEN role = 'Manager' THEN 1000
  WHEN role = 'Sales' THEN 500
  ELSE 0END;
```

 LOOP:

sql

CopyEdit

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

✅Example in PL/SQL:

sql

CopyEdit

```
DECLARE
    v_counter NUMBER := 1;BEGIN
    WHILE v_counter <= 5 LOOP
        DBMS_OUTPUT.PUT_LINE('Counter is: ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;END;
```

Objective: Demonstrate a simple WHILE loop to print numbers 1 to 5.

---

 2. EXCEPTION HANDLING in SQL/PLSQL

 Objective:

Exception handling is used to catch and manage runtime errors (like divide-by-zero, no-data-found, etc.), preventing your program from crashing and allowing graceful recovery or logging.

 Theory:

- Exceptions are handled using the EXCEPTION block in PL/SQL.
- Predefined exceptions: NO_DATA_FOUND, ZERO_DIVIDE, etc.
- You can also define your custom exceptions.

✅Example in PL/SQL:

sql

CopyEdit

```
DECLARE
   v_num1 NUMBER := 10;
   v_num2 NUMBER := 0;
   v_result NUMBER;BEGIN
   v_result := v_num1 / v_num2;
   DBMS_OUTPUT.PUT_LINE('Result is: ' || v_result);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error: Division by zero!');END;
```

Objective: Handle a division-by-zero error gracefully using EXCEPTION.

✅Summary Table:

Concept	Objective	Example/Use
IF / CASE	Conditional logic	Apply bonus based on sales
LOOP / WHILE	Repeat code multiple times	Process records in a loop
EXCEPTION	Handle runtime errors gracefully	Catch divide-by-zero, etc.
EXIT / CONTINUE	Control loop flow	Exit loop early if needed

EXPERIMENT - 9

TITLE :

Create employee management system using MongoDB.

MongoDB is a NoSQL document-oriented database that stores data in JSON-like documents (BSON format). Instead of rows and tables like in relational databases, MongoDB uses collections and documents.

 Employee Management System in MongoDB
We'll cover:
 Database Design
 Collections & Sample Documents
 CRUD Operations (Create, Read, Update, Delete)
 Query Examples

---

1. Database Design
We'll use a database called EmployeeDB.
Collections:
employees – stores employee info

departments – stores department info

attendance – tracks employee check-in/out

salaries – stores salary history

---

2. Sample Documents
 employees collection
json
CopyEdit
```
{
  "_id": ObjectId("..."),
  "emp_id": "E001",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "department_id": "D001",
  "designation": "Software Engineer",
  "hire_date": ISODate("2022-05-01"),
  "status": "active"}
```
 departments collection
json
CopyEdit
```
{
  "_id": ObjectId("..."),
  "dept_id": "D001",
  "name": "Engineering",
```

"manager": "Alice Smith"}

 attendance collection

json

CopyEdit

{

"_id": ObjectId("..."),

"emp_id": "E001",

"date": ISODate("2025-04-13"),

"check_in": "09:00",

"check_out": "17:00"}

 salaries collection

json

CopyEdit

{

"_id": ObjectId("..."),

"emp_id": "E001",

"basic": 60000,

"bonus": 5000,

"month": "April",

"year": 2025}

---

3. Basic CRUD Operations

✅Insert a new employee

js

CopyEdit

```
db.employees.insertOne({
  emp_id: "E002",
  name: "Jane Smith",
  email: "jane.smith@example.com",
  department_id: "D002",
  designation: "HR Manager",
  hire_date: new Date("2023-01-15"),
  status: "active"
});
```

 Find all employees

js

CopyEdit

```
db.employees.find({});
```

 Update employee status

js

CopyEdit

```
db.employees.updateOne(
  { emp_id: "E002" },
  { $set: { status: "inactive" } }
);
```

❌Delete an employee

js

CopyEdit

```
db.employees.deleteOne({ emp_id: "E002" });
```

4. Useful Queries
 Find employees in a specific department
js
CopyEdit
db.employees.find({ department_id: "D001" });
 Get attendance for a specific employee
js
CopyEdit
db.attendance.find({ emp_id: "E001" });
 Calculate total salary paid to an employee for April
js
CopyEdit
db.salaries.aggregate([
  { $match: { emp_id: "E001", month: "April", year: 2025 } },
  {
    $project: {
      total: { $add: ["$basic", "$bonus"] }
    }
  }
]);

✅Summary of Collections and Their Roles

Collection	Purpose
employees	Stores personal and job details
departments	Stores department info
attendance	Logs daily employee activity
salaries	Tracks pay details per employee

EXPERIMENT - 10

TITLE :

Connect employee management system using JDBC.

THEORY:

Using MongoDB Java Driver (Recommended)

This is the most direct and supported way to connect MongoDB with Java.

□ 1. Add MongoDB Driver to Your Project

If using Maven:

xml

CopyEdit

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>mongodb-driver-sync</artifactId>

<version>4.11.0</version></dependency>

□□ 2. Java Code to Connect to MongoDB

java

CopyEdit

import com.mongodb.client.MongoClients;import com.mongodb.client.MongoClient;import com.mongodb.client.MongoDatabase;import com.mongodb.client.MongoCollection;import org.bson.Document;

public class EmployeeManagementSystem {

public static void main(String[] args) {

// Connect to MongoDB

```
        MongoClient mongoClient = MongoClients.create("mongodb://localhost:27017");

        // Access the database

        MongoDatabase database = mongoClient.getDatabase("EmployeeDB");

        // Access a collection

        MongoCollection<Document> employees = database.getCollection("employees");

        // Insert a new employee

        Document emp = new Document("emp_id", "E003")

                .append("name", "Robert Brown")

                .append("email", "robert.brown@example.com")

                .append("department_id", "D001")

                .append("designation", "DevOps Engineer")

                .append("hire_date", "2024-11-01")

                .append("status", "active");


        employees.insertOne(emp);

        System.out.println("Employee inserted successfully.");

    }

}
```

 Advantages of Using MongoDB Java Driver:

- Fast, native support
- Full MongoDB functionality
- Easy to integrate with Java applications

 Option 2: Using MongoDB JDBC Driver (via BI Connector or Unity JDBC)

You can use third-party JDBC drivers like:

MongoDB BI Connector (for SQL over MongoDB)

Unity JDBC Driver for MongoDB

Use Cases:

Integrate MongoDB with tools that require JDBC

Run SQL-like queries on MongoDB from Java

 Sample JDBC-like Connection (via Unity Driver)

java

CopyEdit

```
Class.forName("com.mongodb.jdbc.MongoDriver");Connection conn =
DriverManager.getConnection("jdbc:mongodb://localhost:27017/EmployeeDB");

Statement stmt = conn.createStatement();ResultSet rs = stmt.executeQuery("SELECT name
FROM employees");

while (rs.next()) {

    System.out.println(rs.getString("name"));

}
```

 Note: This only works if you have installed and configured the MongoDB JDBC bridge/driver properly.

 Summary

Method	When to Use
MongoDB Java Driver	✅Recommended for Java applications
JDBC Driver via BI/Unity	For SQL/JDBC tool integration

EXPERIMENT - 11

TITLE :

Executing Queries using NoSQL(document-based).

Theory :

◆ **Executing Queries in NoSQL (Document-Based)**

 What is NoSQL?

NoSQL (Not Only SQL) is a database approach that provides a mechanism for storage and retrieval of data modeled in ways other than the tabular relations used in relational databases. It's designed for distributed data stores with scalability, flexibility, and performance in mind.

 What is Document-Based NoSQL?

Stores data as documents (typically in JSON or BSON format).

Common NoSQL document databases: MongoDB, CouchDB, Firebase Firestore.

Collections hold documents, similar to tables holding rows in relational DBs.

Example Document in MongoDB:

json

CopyEdit

```
{
  "_id": "E001",
  "name": "Alice Smith",
  "department": "HR",
  "email": "alice@example.com",
  "status": "active"}
```

 Objectives of Query Execution in Document-Based NoSQL

Goal	Description
Data Retrieval	Extract data based on conditions (find, filter, etc.)

Goal	Description
□ Data Projection	Return only selected fields from documents
□ Data Manipulation	Perform insert, update, delete operations
□ Aggregation & Grouping	Analyze data using $group, $match, $sum, etc.
□ Performance & Scalability	Optimize data access across distributed nodes

---

□□ Examples of Queries in MongoDB

Assuming a collection called employees.

✅1. Retrieve All Employees

js

CopyEdit

```
db.employees.find({});
```

✅2. Find Employee by Department

js

CopyEdit

```
db.employees.find({ department: "HR" });
```

✅3. Find Employees with Specific Fields Only

js

CopyEdit

```
db.employees.find(
  { status: "active" },
  { name: 1, email: 1, _id: 0 }
);
```

✅4. Insert a New Employee

js

CopyEdit

```
db.employees.insertOne({
  _id: "E002",
  name: "John Doe",
  department: "IT",
  email: "john.doe@example.com",
  status: "active"
});
```

✅5. Update an Employee’s Status

js

CopyEdit

```
db.employees.updateOne(
  { _id: "E002" },
  { $set: { status: "inactive" } }
);
```

✅6. Delete an Employee

js

CopyEdit

```
db.employees.deleteOne({ _id: "E002" });
```

✅7. Count Number of Employees in Each Department

js

CopyEdit

```
db.employees.aggregate([
  { $group: { _id: "$department", total: { $sum: 1 } } }
```

]);

✅8. Search Using Conditions (e.g., multiple filters)

js

CopyEdit

```
db.employees.find({
  department: "IT",
  status: "active"
});
```

 Comparison with SQL (for Clarity)

SQL	MongoDB NoSQL
SELECT * FROM employees;	db.employees.find({});
INSERT INTO employees ...	db.employees.insertOne({...})
UPDATE employees SET ...	db.employees.updateOne(...);
DELETE FROM employees ...	db.employees.deleteOne(...);

 Summary

 Key Objectives:

Efficient and flexible querying of semi-structured data

Simple syntax for CRUD operations

High scalability for large volumes of data

Support for real-time analytics via aggregation framework

 Key Concepts:

- Documents (JSON-like structures)
- Collections (like tables)
- Queries using methods like find(), update(), delete(), and aggregate()

Lab - Assignment on joins:

- Create table named customer table and order table and insert the given data.

```
1  CREATE TABLE Customers (
2      CustomerID INT PRIMARY KEY,
3      Name VARCHAR(50),
4      City VARCHAR(50)
5  );
```

Results | Explain | Describe | Saved SQL | History

Table created.

0.04 seconds

```
4      City VARCHAR(50)
5  );
6  INSERT INTO Customers (CustomerID, Name, City)
7  VALUES
8  (101, 'John', 'New York'),
9  (102, 'Alice', 'Chicago'),
10 (103, 'Bob', 'Los Angeles'),
11 (104, 'Emma', 'Chicago');
```

Results | Explain | Describe | Saved SQL | History

Error at line 3/26: ORA-00933: SQL command not properly ended

```
1. INSERT INTO Customers (CustomerID, Name, City)
2. VALUES
3. (101, 'John', 'New York'),
4. (102, 'Alice', 'Chicago'),
5. (103, 'Bob', 'Los Angeles'),
```

```
(103, 'Bob', 'Los Angeles'),
(104, 'Emma', 'Chicago');
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    Product VARCHAR(50),
    Amount INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE SET NULL
);
```

Results | Explain | Describe | Saved SQL | History

```
Table created.

0.06 seconds
```

```
INSERT INTO Orders (OrderID, CustomerID, Product, Amount)
VALUES
(1, 101, 'Laptop', 50000),
(2, 102, 'Phone', 30000),
(3, 103, 'TV', 40000),
(4, 101, 'Tablet', 20000),
(5, 105, 'Camera', 25000); -- CustomerID 105 does not exist in Customers table
```

Results | Explain | Describe | Saved SQL | History

```
Error at line 3/26: ORA-00933: SQL command not properly ended


1. INSERT INTO Orders (OrderID, CustomerID, Product, Amount)
2. VALUES
3. (1, 101, 'Laptop', 50000),
4. (2, 102, 'Phone', 30000),
5. (3, 103, 'TV', 40000),
```

```
SELECT C.Name, O.Product
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;
```

```
(1, 101, 'Laptop', 50000),
(2, 102, 'Phone', 30000),
(3, 103, 'TV', 40000),
(4, 101, 'Tablet', 20000),
(5, 105, 'Camera', 25000); -- CustomerID 105 does not exist in Customers table
SELECT C.Name, O.Product
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;

SELECT C.Name, O.Product, O.Amount
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID;
```

```sql
(1, 101, 'Laptop', 50000),
(2, 102, 'Phone', 30000),
(3, 103, 'TV', 40000),
(4, 101, 'Tablet', 20000),
(5, 105, 'Camera', 25000); -- CustomerID 105 does not exist in Customers table
SELECT C.Name, O.Product
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;

SELECT C.Name, O.Product, O.Amount
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID;
```

Language SQL Rows 10 Clear command Find tables

```sql
(1, 101, 'Laptop', 50000),
(2, 102, 'Phone', 30000),
(3, 103, 'TV', 40000),
(4, 101, 'Tablet', 20000),
(5, 105, 'Camera', 25000); -- CustomerID 105 does not exist in Customers table
SELECT C.Name, O.Product
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;

SELECT C.Name, O.Product, O.Amount
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID;
SELECT C.Name, O.Product, O.Amount
FROM Customers C
RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID;
```

```sql
SELECT C.Name, O.Product, O.Amount
FROM Customers C
RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID;

SELECT C.Name, O.Product, O.Amount
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
UNION
SELECT C.Name, O.Product, O.Amount
FROM Customers C
RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID;
```

```sql
SELECT C.Name, O.Product, O.Amount
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
UNION
SELECT C.Name, O.Product, O.Amount
FROM Customers C
RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID;

SELECT C1.Name AS Customer1, C2.Name AS Customer2, C1.City
FROM Customers C1
JOIN Customers C2 ON C1.City = C2.City
AND C1.CustomerID < C2.CustomerID;
```

```
38  FROM Customers C
39  LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
40  UNION
41  SELECT C.Name, O.Product, O.Amount
42  FROM Customers C
43  RIGHT JOIN Orders O ON C.CustomerID = O.CustomerID;
44
45  SELECT C1.Name AS Customer1, C2.Name AS Customer2, C1.City
46  FROM Customers C1
47  JOIN Customers C2 ON C1.City = C2.City
48  AND C1.CustomerID < C2.CustomerID;
49
50  SELECT C.Name, COALESCE(SUM(O.Amount), 0) AS TotalAmount
51  FROM Customers C
52  LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
53  GROUP BY C.Name;
54
```

Lab - Assignment on Sub Queries :

1□
```
CREATE TABLE Employees (
id INT PRIMARY KEY,
name VARCHAR(100),
salary DECIMAL(10,2),
department_id INT,
join_date DATE
);
```

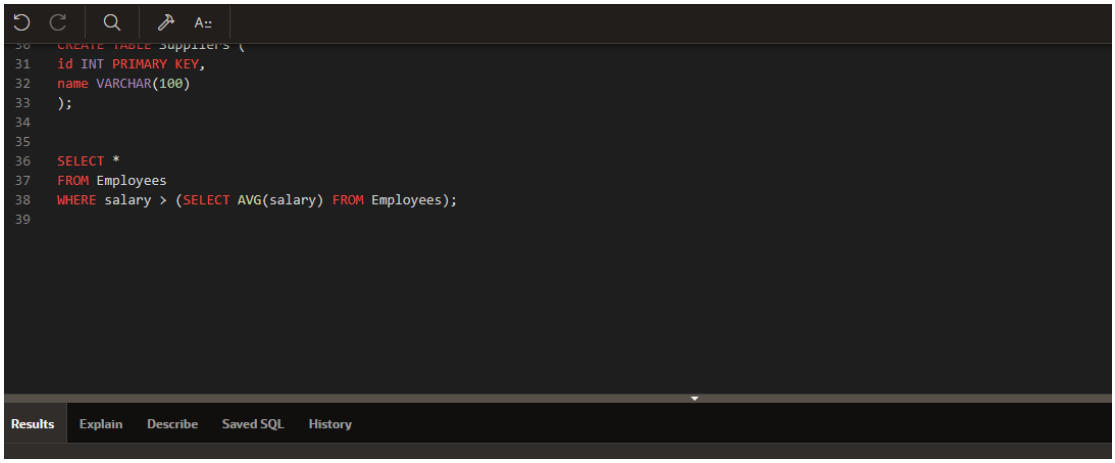
2□
```
CREATE TABLE Departments (
id INT PRIMARY KEY,
name VARCHAR(100)
);
```

3□
```
CREATE TABLE Orders (
id INT PRIMARY KEY,
customer_id INT,
order_date DATE,
total_amount DECIMAL(10,2)
);
```

4□
```
CREATE TABLE Customers (
id INT PRIMARY KEY,
name VARCHAR(100),
city VARCHAR(100)
);
```

5□
```
CREATE TABLE Products (
id INT PRIMARY KEY,
name VARCHAR(100),
category VARCHAR(50),
price DECIMAL(10,2),
supplier_id INT
);
```

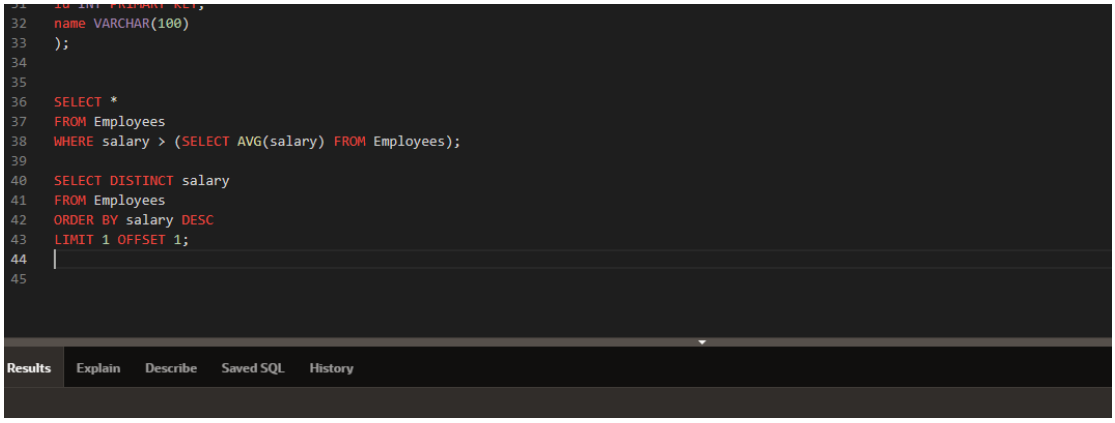
6□
```
CREATE TABLE Suppliers (
id INT PRIMARY KEY,
name VARCHAR(100)
);
```



```
CREATE TABLE Suppliers (
id INT PRIMARY KEY,
name VARCHAR(100)
);


SELECT *
FROM Employees
WHERE salary > (SELECT AVG(salary) FROM Employees);
```

Results Explain Describe Saved SQL History

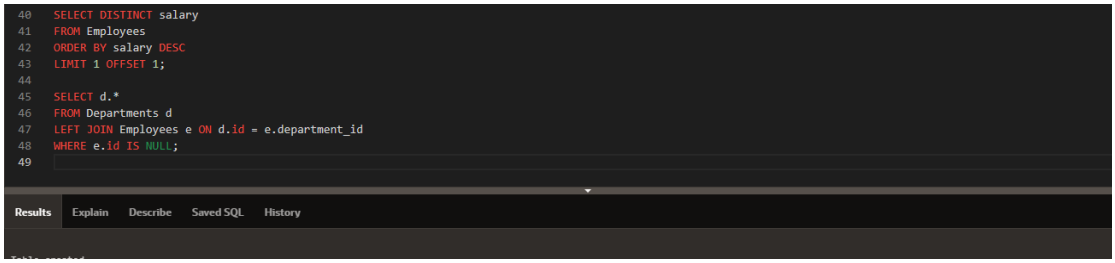


```
id INT PRIMARY KEY,
name VARCHAR(100)
);


SELECT *
FROM Employees
WHERE salary > (SELECT AVG(salary) FROM Employees);

SELECT DISTINCT salary
FROM Employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

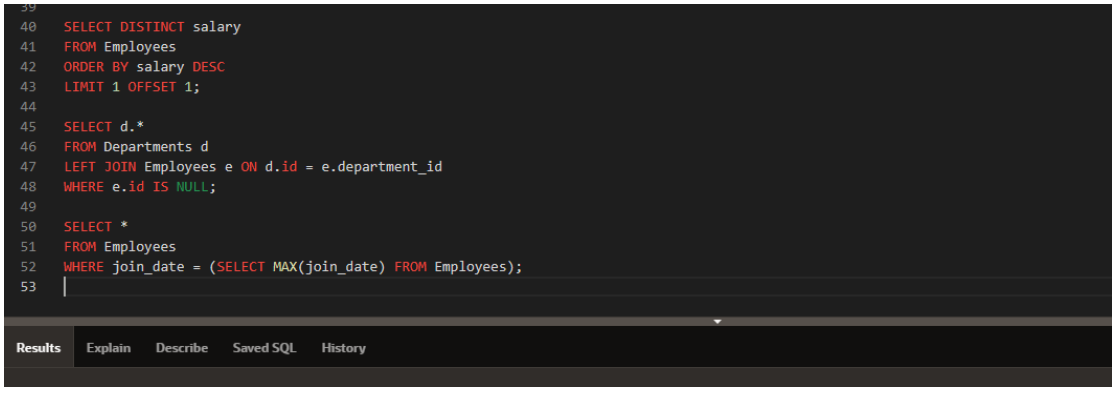
Results Explain Describe Saved SQL History



```
SELECT DISTINCT salary
FROM Employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;

SELECT d.*
FROM Departments d
LEFT JOIN Employees e ON d.id = e.department_id
WHERE e.id IS NULL;
```

Results Explain Describe Saved SQL History



```
SELECT DISTINCT salary
FROM Employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;

SELECT d.*
FROM Departments d
LEFT JOIN Employees e ON d.id = e.department_id
WHERE e.id IS NULL;

SELECT *
FROM Employees
WHERE join_date = (SELECT MAX(join_date) FROM Employees);
```

Results Explain Describe Saved SQL History

```
44
45  SELECT d.*
46  FROM Departments d
47  LEFT JOIN Employees e ON d.id = e.department_id
48  WHERE e.id IS NULL;
49
50  SELECT *
51  FROM Employees
52  WHERE join_date = (SELECT MAX(join_date) FROM Employees);
53
54  SELECT *
55  FROM Products
56  WHERE id IN (SELECT product_id FROM OrderDetails);
57
```

Results | Explain | Describe | Saved SQL | History

```
49
50  SELECT *
51  FROM Employees
52  WHERE join_date = (SELECT MAX(join_date) FROM Employees);
53
54  SELECT *
55  FROM Products
56  WHERE id IN (SELECT product_id FROM OrderDetails);
57
58  SELECT *
59  FROM Employees
60  WHERE join_date = (SELECT MAX(join_date) FROM Employees);
61
```

Results | Explain | Describe | Saved SQL | History

```
56  WHERE id IN (SELECT product_id FROM OrderDetails);
57
58  SELECT *
59  FROM Employees
60  WHERE join_date = (SELECT MAX(join_date) FROM Employees);
61
62  SELECT e.*
63  FROM Employees e
64  WHERE e.salary > (
65      SELECT AVG(salary)
66      FROM Employees
67      WHERE department_id = e.department_id
68  );
69
```

Results | Explain | Describe | Saved SQL | History

```
65      SELECT AVG(salary)
66      FROM Employees
67      WHERE department_id = e.department_id
68  );
69  SELECT department_id, COUNT(*) AS employee_count
70  FROM Employees
71  GROUP BY department_id
72  ORDER BY employee_count DESC
73  LIMIT 1;
74
```

Results | Explain | Describe | Saved SQL | History

```
    SELECT AVG(salary)
    FROM Employees
    WHERE department_id = e.department_id
);
SELECT department_id, COUNT(*) AS employee_count
FROM Employees
GROUP BY department_id
ORDER BY employee_count DESC
LIMIT 1;

SELECT c.*
FROM Customers c
LEFT JOIN Orders o ON c.id = o.customer_id
WHERE o.id IS NULL;
```

Results | Explain | Describe | Saved SQL | History

```
SELECT c.*
FROM Customers c
LEFT JOIN Orders o ON c.id = o.customer_id
WHERE o.id IS NULL;

SELECT *
FROM Products
WHERE price > (
    SELECT MAX(price)
    FROM Products
    WHERE category = 'Electronics'
);
```

Results | Explain | Describe | Saved SQL | History

```
    FROM Products
    WHERE category = 'Electronics'
);
SELECT *
FROM Orders
WHERE total_amount = (SELECT MAX(total_amount) FROM Orders);
SELECT s.*
FROM Suppliers s
WHERE EXISTS (
    SELECT 1
    FROM Products p
    WHERE p.supplier_id = s.id AND p.price > 500
);
```

Results | Explain | Describe | Saved SQL | History

Lab - Assignment on Views

Q.1 Create a view detailsview having sid,name,address using table studentdetail.

```
CREATE VIEW detailsview AS
SELECT sid, name, address
FROM studentdetail;
```

Q.2 Write a query to display the record from detail-view.

```
SELECT * FROM detailsview;
```

Q.3 Write a query to drop detailview.

```
DROP VIEW detailsview;
```

Q.4 Write a query to create view from multiple tables.

```
CREATE VIEW student_marks_view AS
SELECT sd.sid, sd.name, m.subject, m.marks
FROM studentdetail sd
JOIN marks m ON sd.sid = m.sid;
```

Q.5 Write a query to display the record from marks view.

```
SELECT * FROM marksview;
```

Q.6 Write a query to update the view

```
UPDATE marksview
SET marks = 95
WHERE sid = 101 AND subject = 'Math';
```

Q.7 Write a query to display the updated marks view.

```
SELECT * FROM marksview;
```

Lab - Assignment On Views , Set Operation

The college database contains the following tables:

- Students(student_id, student_name, department_id, age)
- Departments(department_id, department_name)
- Courses(course_id, course_name, department_id)
- Enrollments(enrollment_id, student_id, course_id, marks)
- Professors(professor_id, professor_name, department_id)
- Teaches(professor_id, course_id)

ANS :

CREATING TABLES :

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    department_id INT,
    age INT,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id)
);

CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100) NOT NULL
);

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id)
```

Results | Explain | Describe | Saved SQL | History

```
Table created.
```

```sql
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id)
);

CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    marks INT CHECK (marks BETWEEN 0 AND 100),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

CREATE TABLE Professors (
    professor id INT PRIMARY KEY,
```

Results | Explain | Describe | Saved SQL | History

Table created.

```sql
CREATE TABLE Professors (
    professor_id INT PRIMARY KEY,
    professor_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id)
);

CREATE TABLE Teaches (
    professor_id INT,
    course_id INT,
    PRIMARY KEY (professor_id, course_id),
    FOREIGN KEY (professor_id) REFERENCES Professors(professor_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

Results | Explain | Describe | Saved SQL | History

Table created.

Q1: List All Students Along with Their Department Names (INNER JOIN)

```sql
    FOREIGN KEY (professor_id) REFERENCES Professors(professor_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
SELECT s.student_id, s.student_name, d.department_name
FROM Students s
INNER JOIN Departments d ON s.department_id = d.department_id;
```

Q2: Find Students Who Have Not Enrolled in Any Course (LEFT JOIN)

```
INNER JOIN Departments d ON s.department_id = d.department_id;

SELECT s.student_id, s.student_name
FROM Students s
LEFT JOIN Enrollments e ON s.student_id = e.student_id
WHERE e.enrollment_id IS NULL;
```

Q3: Retrieve the Name of the Student Who Scored the Highest Marks in Any Course (Subquery)

```
52
53  SELECT s.student_name
54  FROM Students s
55  WHERE s.student_id = (
56      SELECT student_id
57      FROM Enrollments
58      ORDER BY marks DESC
59      LIMIT 1
60  );
61
```

Q4: Find the Students Who Have Taken Courses in More Than One Department (Subquery with HAVING)Set Operations

```
60  );
61
62  SELECT e.student_id, s.student_name
63  FROM Enrollments e
64  JOIN Courses c ON e.course_id = c.course_id
65  JOIN Students s ON e.student_id = s.student_id
66  GROUP BY e.student_id, s.student_name
67  HAVING COUNT(DISTINCT c.department_id) > 1;
68
```

Q5: List All Student and Professor Names (UNION)

```
67  HAVING COUNT(DISTINCT c.department_id) > 1;
68
69  SELECT student_name AS name FROM Students
70  UNION
71  SELECT professor_name FROM Professors;
72
```

Q6: Find Students Who Are Also Professors (INTERSECT)

```
71  SELECT professor_name FROM Professors;
72
73  SELECT student_name FROM Students
74  INTERSECT
75  SELECT professor_name FROM Professors;
76
77
```

Q7: List Students Who Have Not Taken Any Course (EXCEPT)

```
76
77  SELECT student_name FROM Students
78  EXCEPT
79  SELECT DISTINCT s.student_name
80  FROM Students s
81  JOIN Enrollments e ON s.student_id = e.student_id;
```

Q8: Views and Views Based on ViewsQ8: Create a View for Students Who Have Scored More Than 80 Marks

```
82
83  CREATE VIEW HighScoringStudents AS
84  SELECT s.student_id, s.student_name, e.marks
85  FROM Students s
86  JOIN Enrollments e ON s.student_id = e.student_id
87  WHERE e.marks > 80;
88
89  CREATE VIEW TopScoringStudents AS
```

Q9: Create a View Based on HighScoringStudents for Students Who Have Scored Above 90

```
88
89  CREATE VIEW TopScoringStudents AS
90  SELECT * FROM HighScoringStudents
91  WHERE marks > 90;
92
```

Lab - Assignment on PL/SQL

1. Write a PL/SQL program to arrange the number of two variable in such a way that the small number will store in num_small variable and large number will store in num_large variable.

Ans :

```
Declare
    A   Number := 90;
    B   NUmber := 45;
    num_small Number ;
    num_large Number ;
Begin
    If A > B then
        num_small := B;
        num_large := A;
    Else
        num_small := A;
        num_large := B;
    End if ;
    DBMS_OUTPUT.PUT_LINE('Small number: ' || num_small);
    DBMS_OUTPUT.PUT_LINE('Large number: ' || num_large);
END;
```

Results | Explain | Describe | Saved SQL | History

```
Small number: 45
Large number: 90

Statement processed.
```

2. Write a PL/SQL program to check whether a number is even or odd.

Ans:

```
DECLARE
    num NUMBER := 15;  -- You can change this value to test other numbers
BEGIN
    IF MOD(num, 2) = 0 THEN
        DBMS_OUTPUT.PUT_LINE(num || ' is Even.');
    ELSE
        DBMS_OUTPUT.PUT_LINE(num || ' is Odd.');
    END IF;
END;
```

Results | Explain | Describe | Saved SQL | History

```
15 is Odd.

Statement processed.
```

3. Write a PL/SQL program to display the description against a grade.

Ans:

```
DECLARE
    grade       CHAR(1) := 'C';
    description VARCHAR2(20);
BEGIN
    CASE grade
        WHEN 'A' THEN
            description := 'Excellent';
        WHEN 'B' THEN
            description := 'Very Good';
        WHEN 'C' THEN
            description := 'Good';
        WHEN 'D' THEN
            description := 'Fair';
        WHEN 'E' THEN
            description := 'Poor';
        WHEN 'F' THEN
            description := 'Fail';
        ELSE
            description := 'Invalid Grade';
    END CASE;

    DBMS_OUTPUT.PUT_LINE('Grade: ' || grade);
    DBMS_OUTPUT.PUT_LINE('Description: ' || description);
END;
```

Results | Explain | Describe | Saved SQL | History

```
Grade: B
Description: Very Good

Statement processed.
```

4.Write a PL/SQL program to display the description against a grade using CASE statement

Ans:

```
DECLARE
    grade       CHAR(1) := 'A';  -- Change this value to test different grades
    description VARCHAR2(20);
BEGIN
    description := CASE grade
        WHEN 'A' THEN 'Excellent'
        WHEN 'B' THEN 'Very Good'
        WHEN 'C' THEN 'Good'
        WHEN 'D' THEN 'Fair'
        WHEN 'E' THEN 'Poor'
        WHEN 'F' THEN 'Fail'
        ELSE 'Invalid Grade'
    END;

    DBMS_OUTPUT.PUT_LINE('Grade: ' || grade);
    DBMS_OUTPUT.PUT_LINE('Description: ' || description);
END;
```

Results | Explain | Describe | Saved SQL | History

```
Grade: B
Description: Very Good

Statement processed.
```

Lab - Assignment on PL/SQL loop

- DECLARE
  i number(1);

  j number(1);

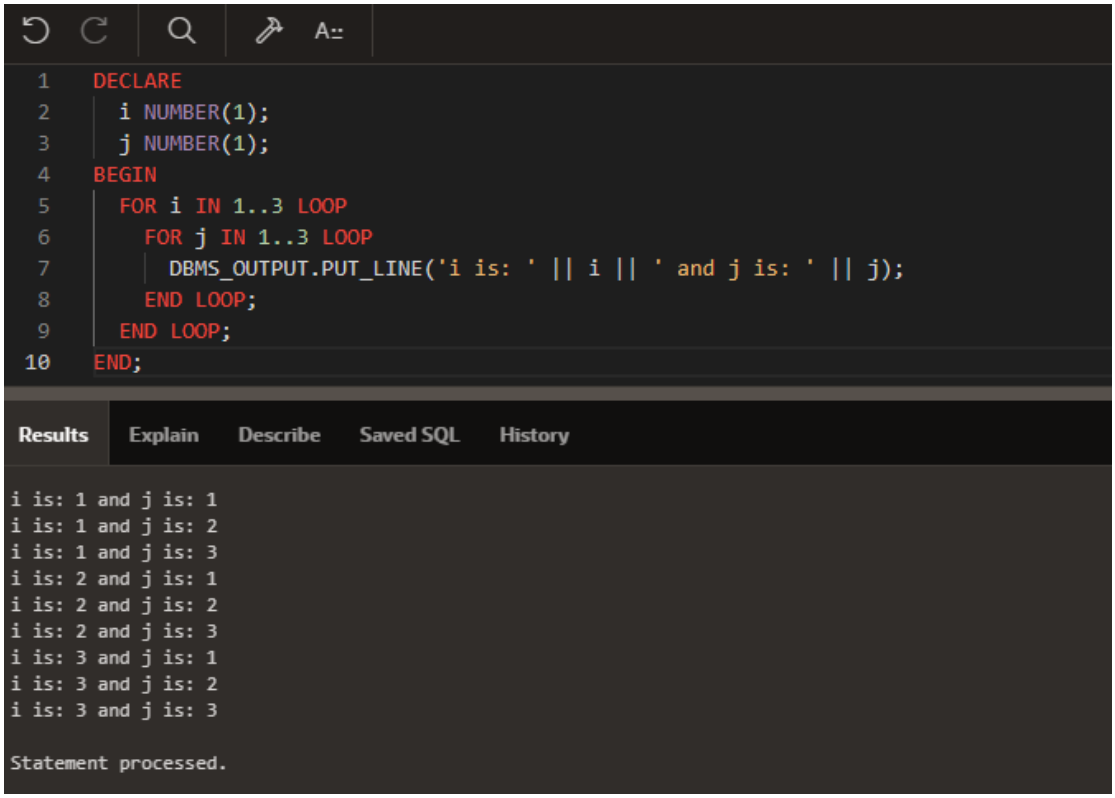
  BEGIN
  FOR i IN 1..3 LOOP

  FOR j IN 1..3 LOOP
  dbms_output.put_line('i is: '|| i || ' and j is: ' || j);

  END loop;

  END loop ;

  END;
  /



- DECLARE
  c_id customers.id%type := 8;
  c_name customerS.Name%type;
  c_addr customers.address%type;
-

  BEGIN
  SELECT name, address INTO c_name, c_addr
  FROM customers
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  EXCEPTION
  WHEN no_data_found THEN
  dbms_output.put_line('No such customer!');
  WHEN others THEN
  dbms_output.put_line('Error!');
  END;
  /

```
12
13  DECLARE
14    c_id    customers.id%TYPE := 8;
15    c_name  customers.name%TYPE;
16    c_addr  customers.address%TYPE;
17  BEGIN
18    SELECT name, address INTO c_name, c_addr
19    FROM customers
20    WHERE id = c_id;
21
22    DBMS_OUTPUT.PUT_LINE('Name: ' || c_name);
23    DBMS_OUTPUT.PUT_LINE('Address: ' || c_addr);
24
25  EXCEPTION
26    WHEN NO_DATA_FOUND THEN
27      DBMS_OUTPUT.PUT_LINE('No such customer!');
28    WHEN OTHERS THEN
29      DBMS_OUTPUT.PUT_LINE('Error!');
30  END;
31
```

Results | Explain | Describe | Saved SQL | History