Practical 1

Aim: Comparative study of various Database Management Systems

A comparative study of various Database Management Systems (DBMS) involves evaluating different systems based on various criteria such as performance, scalability, ease of use, features, security, and cost. Here, I'll provide a general overview of some popular DBMS and highlight key aspects for comparison:

Relational Database Management Systems (RDBMS):

- MySQL: An open-source RDBMS known for its speed and reliability. It's widely used in web applications.
- PostgreSQL: Another open-source RDBMS with a strong emphasis on standards compliance and extensibility.
- Oracle Database: A commercial RDBMS known for its robustness, scalability, and comprehensive feature set.

NoSQL Databases:

- MongoDB (Document Store): A popular NoSQL database that stores data in flexible, JSON-like documents. Suitable for handling unstructured data.
- Cassandra (Wide Column Store): Designed for handling large amounts of distributed data across commodity servers. It's highly scalable and provides high availability.
- Redis (Key-Value Store): An in-memory data structure store, often used as a caching mechanism due to its speed.

NewSQL Databases:

- Google Spanner: A globally distributed, horizontally scalable, and strongly consistent database service. Suitable for applications with global reach.
- CockroachDB: An open-source distributed SQL database that provides consistency, scalability, and survivability across multiple nodes.

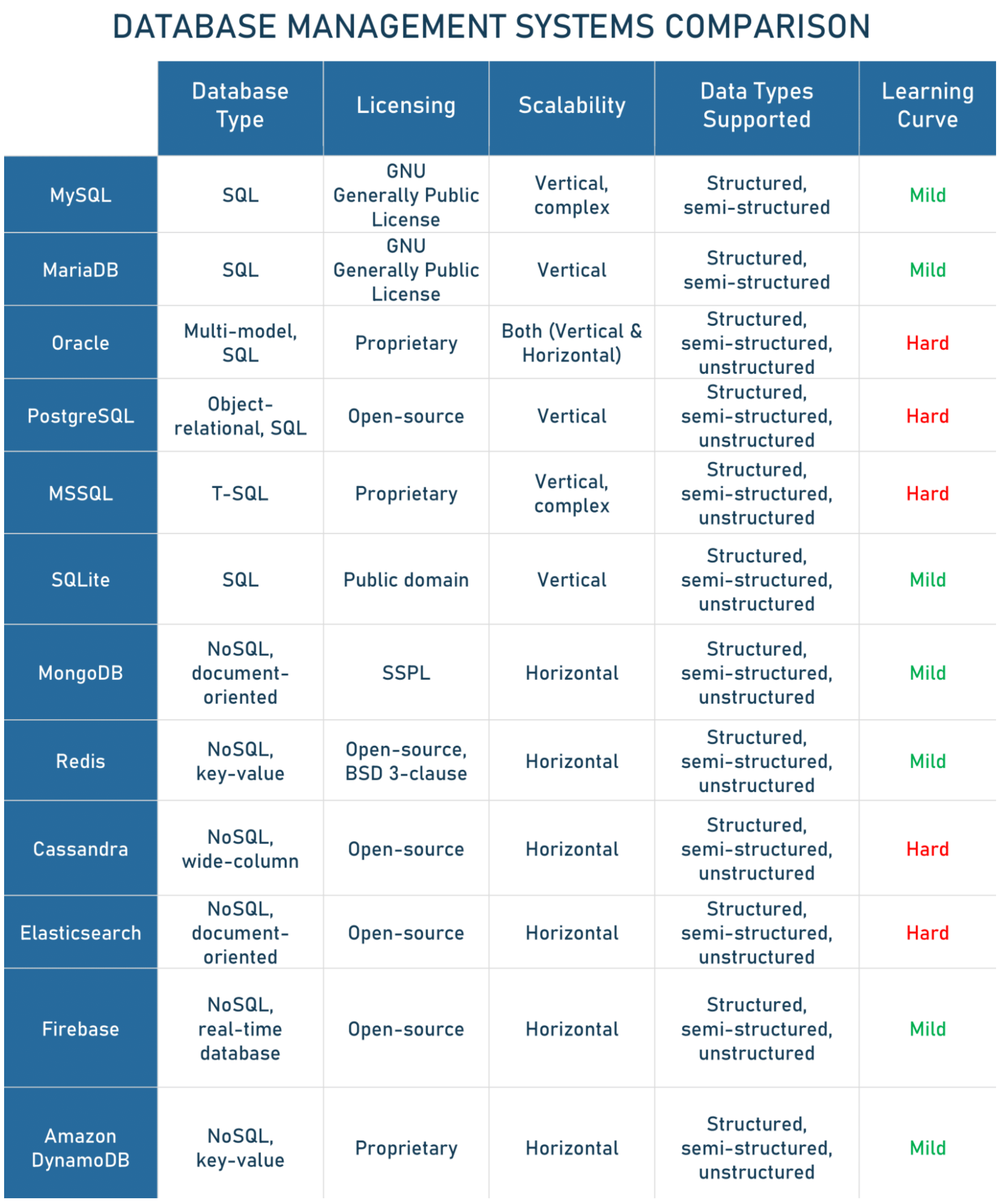
Graph Databases:

- Neo4j: A graph database designed for handling highly interconnected data, making it suitable for applications like social networks and recommendation engines.

Time Series Databases:

- InfluxDB: Designed for handling time-series data, commonly used in monitoring and analytics applications.

Criteria for Comparison:



DATABASE MANAGEMENT SYSTEMS COMPARISON

	Database Type	Licensing	Scalability	Data Types Supported	Learning Curve
MySQL	SQL	GNU Generally Public License	Vertical, complex	Structured, semi-structured	Mild
MariaDB	SQL	GNU Generally Public License	Vertical	Structured, semi-structured	Mild
Oracle	Multi-model, SQL	Proprietary	Both (Vertical & Horizontal)	Structured, semi-structured, unstructured	Hard
PostgreSQL	Object-relational, SQL	Open-source	Vertical	Structured, semi-structured, unstructured	Hard
MSSQL	T-SQL	Proprietary	Vertical, complex	Structured, semi-structured, unstructured	Hard
SQLite	SQL	Public domain	Vertical	Structured, semi-structured, unstructured	Mild
MongoDB	NoSQL, document-oriented	SSPL	Horizontal	Structured, semi-structured, unstructured	Mild
Redis	NoSQL, key-value	Open-source, BSD 3-clause	Horizontal	Structured, semi-structured, unstructured	Mild
Cassandra	NoSQL, wide-column	Open-source	Horizontal	Structured, semi-structured, unstructured	Hard
Elasticsearch	NoSQL, document-oriented	Open-source	Horizontal	Structured, semi-structured, unstructured	Hard
Firebase	NoSQL, real-time database	Open-source	Horizontal	Structured, semi-structured, unstructured	Mild
Amazon DynamoDB	NoSQL, key-value	Proprietary	Horizontal	Structured, semi-structured, unstructured	Mild

Practical 2

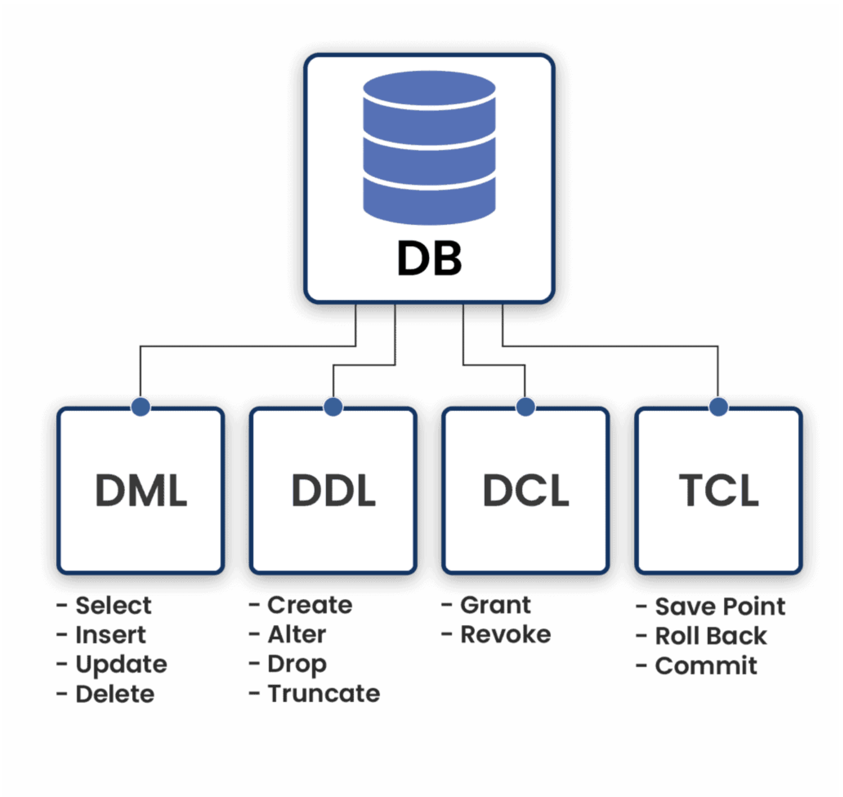
Aim: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL)

SQL commands are like instructions to a table. It is used to interact with the database with some operations. It is also used to perform specific tasks, functions, and queries of data. SQL can perform various tasks like creating a table, adding data to tables, dropping the table, modifying the table, set permission for users.

These SQL commands are mainly categorised into five categories:

- DDL – Data Definition Language
- DML – Data Manipulation Language
- DCL – Data Control Language

Now, we will see all of these in detail.



DDL (Data Definition Language):

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. DDL is a set of SQL commands used to create, modify, and delete database structures but not data. These commands are normally not used by a general user, who should be accessing the database via an application.

List of DDL commands:

- CREATE: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- DROP: This command is used to delete objects from the database.
- ALTER: This is used to alter the structure of the database.
- TRUNCATE: This is used to remove all records from a table, including all spaces allocated for the records are removed.
- COMMENT: This is used to add comments to the data dictionary.
- RENAME: This is used to rename an object existing in the database.

DML(Data Manipulation Language):

The SQL commands that deal with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

List of DML commands:

- INSERT: It is used to insert data into a table.
- UPDATE: It is used to update existing data within a table.
- DELETE: It is used to delete records from a database table.
- LOCK: Table control concurrency.
- CALL: Call a PL/SQL or JAVA subprogram.
- EXPLAIN PLAN: It describes the access path to data.

DCL (Data Control Language)
DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.
List of DCL commands:

- GRANT: This command gives users access privileges to the database.

  Syntax: GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

- REVOKE: This command withdraws the user's access privileges given by using the GRANT command.

  Syntax: REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

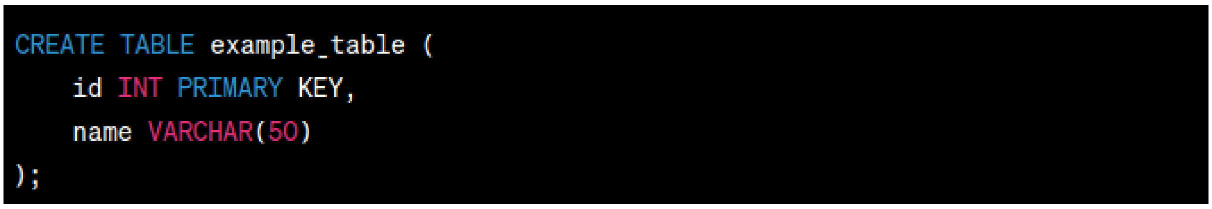
Practical 3

Aim: How to apply Constraints at various levels.

In a relational database, constraints are rules that are applied to the data in tables to maintain the integrity and consistency of the data. Constraints can be applied at various levels, including at the column level, table level, and database level. Here's how you can apply constraints at each level:

1. Column-Level Constraints:

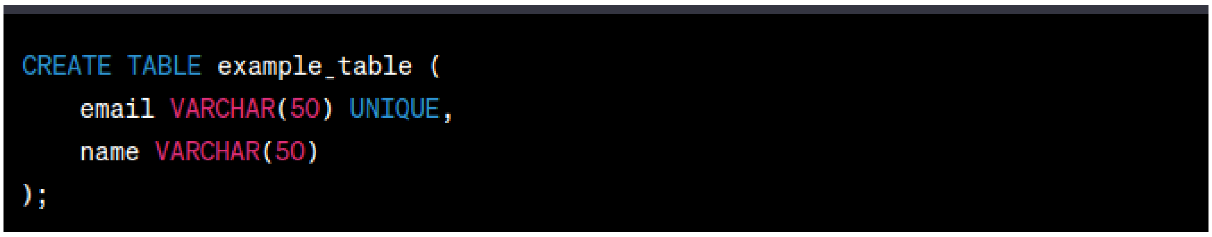
Primary Key Constraint:



```
CREATE TABLE example_table (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

In this example, the ‘id’ column is defined as the primary key, which implies that it must contain unique values, and it cannot be NULL.

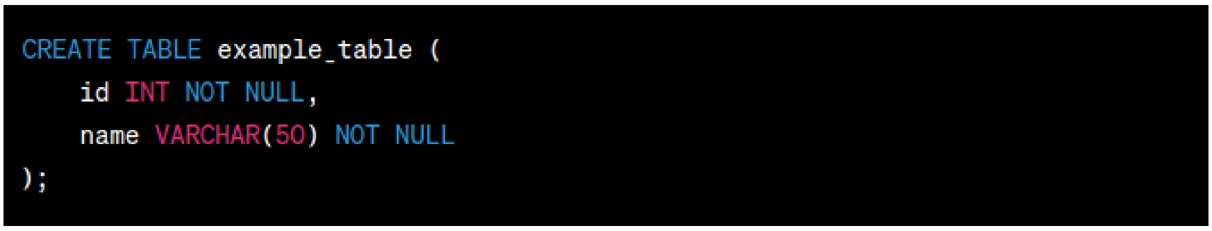
Unique Constraint:



```
CREATE TABLE example_table (
    email VARCHAR(50) UNIQUE,
    name VARCHAR(50)
);
```

The ‘email’ column has a unique constraint, ensuring that each value in this column is unique across all rows in the table.

Not Null Constraint:



```
CREATE TABLE example_table (
    id INT NOT NULL,
    name VARCHAR(50) NOT NULL
);
```

The id and name columns cannot contain NULL values.

2. Table-Level Constraints:

Foreign Key Constraint:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Here, the orders table has a foreign key constraint (product_id) that references the primary key (product_id) in the products table.

Check Constraint:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    salary DECIMAL(10, 2),
    CHECK (salary > 0)
);
```

The CHECK constraint ensures that the salary column must be greater than 0 for every row in the employees table.

3. Database-Level Constraints:

In some database systems, you can define constraints that span multiple tables. However, the syntax for defining such constraints may vary between database management systems.

```
-- Create a table for products
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    price DECIMAL(10, 2)
);

-- Create a table for orders
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    quantity INT,
    total_price DECIMAL(10, 2),
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Practical 4

Aim: View data in the required form using Operators, Functions and Joins.

To view data in a required form using Operators, Functions, and Joins, we'll create an example scenario with two tables: employees and departments. We'll use various SQL operations to retrieve and present the data in a meaningful way.

Assume the following table structures:

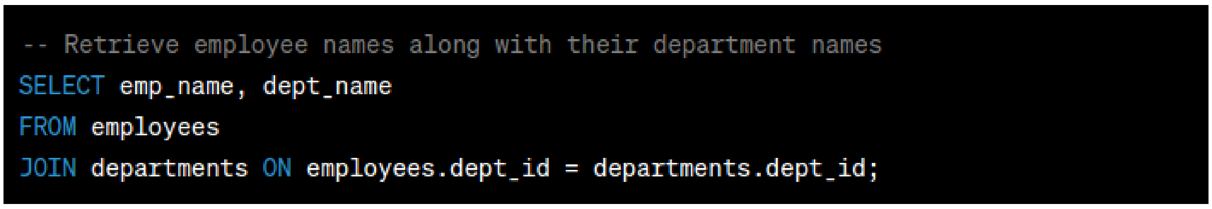
```
-- Employees table
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    emp_salary DECIMAL(10, 2),
    dept_id INT
);

-- Departments table
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

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

```
-- Insert sample data into employees table
INSERT INTO employees (emp_id, emp_name, emp_salary, dept_id) VALUES
(1, 'John Doe', 50000, 1),
(2, 'Jane Smith', 60000, 1),
(3, 'Bob Johnson', 55000, 2),
(4, 'Alice Brown', 62000, 2);

-- Insert sample data into departments table
INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'Sales'),
(2, 'Marketing');
```

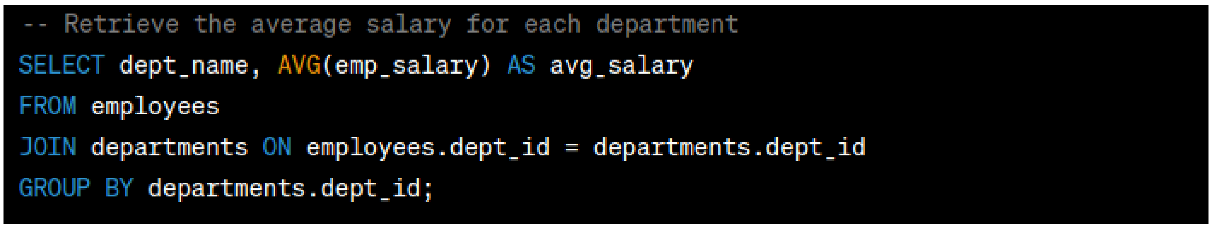
Now, let's use Operators, Functions, and Joins to view the data in a required form:

1. Selecting Data with Joins:



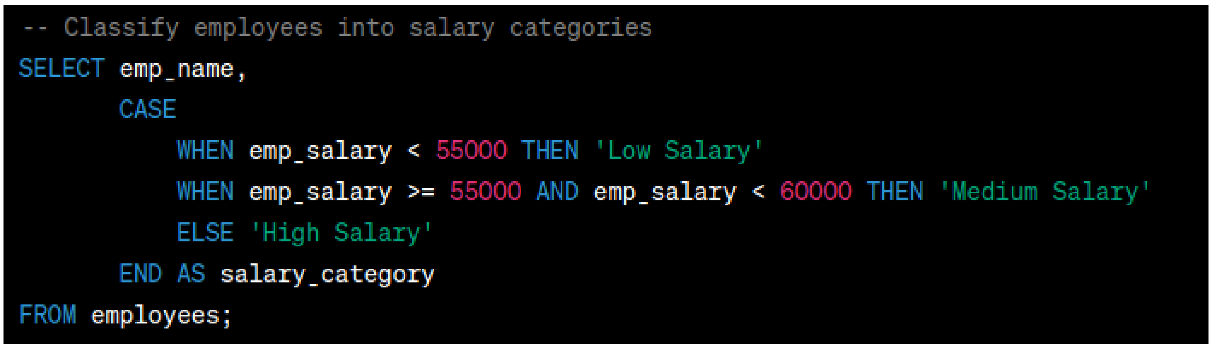
```sql
-- Retrieve employee names along with their department names
SELECT emp_name, dept_name
FROM employees
JOIN departments ON employees.dept_id = departments.dept_id;
```

2. Using Aggregate Functions:



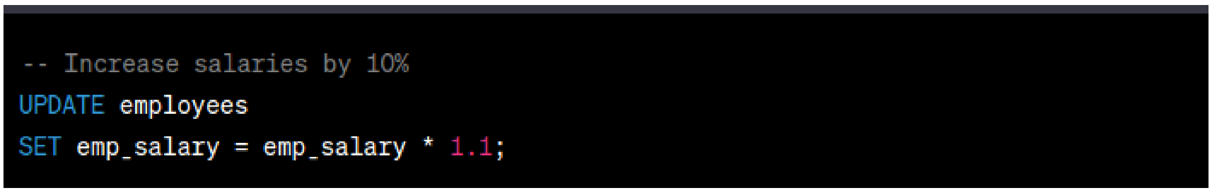
```sql
-- Retrieve the average salary for each department
SELECT dept_name, AVG(emp_salary) AS avg_salary
FROM employees
JOIN departments ON employees.dept_id = departments.dept_id
GROUP BY departments.dept_id;
```

3. Applying Conditional Logic with CASE Statement:



```sql
-- Classify employees into salary categories
SELECT emp_name,
       CASE
           WHEN emp_salary < 55000 THEN 'Low Salary'
           WHEN emp_salary >= 55000 AND emp_salary < 60000 THEN 'Medium Salary'
           ELSE 'High Salary'
       END AS salary_category
FROM employees;
```

4. Using Arithmetic Operators:



```sql
-- Increase salaries by 10%
UPDATE employees
SET emp_salary = emp_salary * 1.1;
```

5. Combining Multiple Operations:

```
-- Retrieve employee names, department names, and total salary for each employee
SELECT emp_name, dept_name, emp_salary,
       emp_salary + COALESCE((SELECT SUM(emp_salary) FROM employees WHERE
dept_id = e.dept_id AND emp_id != e.emp_id), 0) AS total_salary
FROM employees e
JOIN departments ON e.dept_id = departments.dept_id;
```

Practical 5

Aim: Creating different types of Views for tailored presentation of data

Creating views in a relational database allows you to present data in a tailored way without modifying the underlying tables. Views can be used to simplify complex queries, provide a layer of security, and offer a convenient way to present specific data to users. Below are examples of creating different types of views in MySQL:

Assuming you have tables named employees and departments as defined in the previous examples:

1. Basic View:

A basic view can be created to display information from one or more tables.

```sql
-- Create a basic view to display employee names and their salaries
CREATE VIEW employee_salaries AS
SELECT emp_name, emp_salary
FROM employees;
```

2. Joining Tables in a View:

You can create a view that joins multiple tables for a more comprehensive presentation.

```sql
-- Create a view to display employee names, their departments, and department names
CREATE VIEW employee_departments AS
SELECT emp_name, emp_salary, dept_name
FROM employees
JOIN departments ON employees.dept_id = departments.dept_id;
```

3. Aggregated View:

Create a view that provides aggregated information, such as the average salary for each department.

```sql
-- Create a view to display average salary for each department
CREATE VIEW avg_salary_by_department AS
SELECT dept_name, AVG(emp_salary) AS avg_salary
FROM employees
JOIN departments ON employees.dept_id = departments.dept_id
GROUP BY departments.dept_id;
```

4. Conditional View:

Create a view that presents data based on certain conditions.

```sql
-- Create a view to display employees with high salaries
CREATE VIEW high_salary_employees AS
SELECT emp_name, emp_salary
FROM employees
WHERE emp_salary > 60000;
```

5. Updatable View:

In some cases, you can create updatable views. However, there are restrictions on what can be updated. This example shows a simple updatable view:

```sql
-- Create an updatable view to display and update employee names
CREATE VIEW updatable_employee_names AS
SELECT emp_id, emp_name
FROM employees
WHERE emp_salary > 50000;
```

6. Read-Only View:

Create a view that is explicitly read-only to prevent any modifications.

```sql
-- Create a read-only view to display employee names and departments
CREATE VIEW readonly_employee_departments AS
SELECT emp_name, dept_name
FROM employees
JOIN departments ON employees.dept_id = departments.dept_id
WITH CHECK OPTION;
```

Practical 6

Aim: How to apply Conditional Controls in PL/SQL

In PL/SQL (Procedural Language/Structured Query Language), you can use conditional control structures to implement decision-making in your code. These structures allow you to execute different blocks of code based on specified conditions. The primary conditional control structures in PL/SQL are IF-THEN-ELSE and CASE.

1. IF-THEN-ELSE:

The IF-THEN-ELSE statement is used to execute different blocks of code based on a specified condition.

```
DECLARE
 x NUMBER := 10;
BEGIN
 IF x > 5 THEN
 DBMS_OUTPUT.PUT_LINE('x is greater than 5');
 ELSE
 DBMS_OUTPUT.PUT_LINE('x is not greater than 5');
 END IF;
END;
```

In this example, the code checks whether the value of x is greater than 5. If the condition is true, it executes the code inside the THEN block; otherwise, it executes the code inside the ELSE block.

2. CASE Statement:

The CASE statement is used to perform conditional logic similar to switch statements in other programming languages.

```
DECLARE
 day_of_week NUMBER := 3;
 day_name VARCHAR2(20);
BEGIN
 CASE day_of_week
 WHEN 1 THEN
 day_name := 'Sunday';
 WHEN 2 THEN
 day_name := 'Monday';
 WHEN 3 THEN
 day_name := 'Tuesday';
 WHEN 4 THEN
 day_name := 'Wednesday';
 WHEN 5 THEN
 day_name := 'Thursday';
 WHEN 6 THEN
 day_name := 'Friday';
 WHEN 7 THEN
 day_name := 'Saturday';
 ELSE
 day_name := 'Invalid day';
 END CASE;

 DBMS_OUTPUT.PUT_LINE('Day of the week: ' || day_name);
END;
```

In this example, the CASE statement checks the value of day_of_week and assigns the corresponding day name to the day_name variable.

3. Nested IF-THEN-ELSE:

You can nest IF-THEN-ELSE statements to create more complex conditional logic.

```
DECLARE
 x NUMBER := 10;
 y NUMBER := 20;
BEGIN
 IF x > 5 THEN
 IF y > 15 THEN
  DBMS_OUTPUT.PUT_LINE('Both x and y are greater than their respective
thresholds');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Only x is greater than its threshold');
 END IF;
 ELSE
 DBMS_OUTPUT.PUT_LINE('x is not greater than 5');
 END IF;
END;
```

In this example, the code first checks if x is greater than 5. If true, it then checks if y is greater than 15.

These examples illustrate how to use conditional controls in PL/SQL. Depending on the complexity of your conditions, you can use combinations of these structures to achieve the desired logic flow.

Practical 7

Aim: Error Handling using Internal Exceptions and External Exceptions.

In PL/SQL, error handling is crucial for managing unexpected issues that may occur during the execution of a block of code. PL/SQL provides mechanisms for handling both internal and external exceptions. Here's an overview of how to handle errors using internal and external exceptions:

Internal Exceptions:

Internal exceptions are predefined errors raised by the Oracle Database system. PL/SQL provides a set of predefined exceptions that you can use to handle common errors. The EXCEPTION block is used to handle internal exceptions.

```
DECLARE
    x NUMBER := 0;
BEGIN
    -- Attempting to divide by zero
    DBMS_OUTPUT.PUT_LINE(10 / x);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Cannot divide by zero');
END;
/
```

In this example, if the variable x is 0, a ZERO_DIVIDE exception will be raised, and the code inside the EXCEPTION block will be executed.

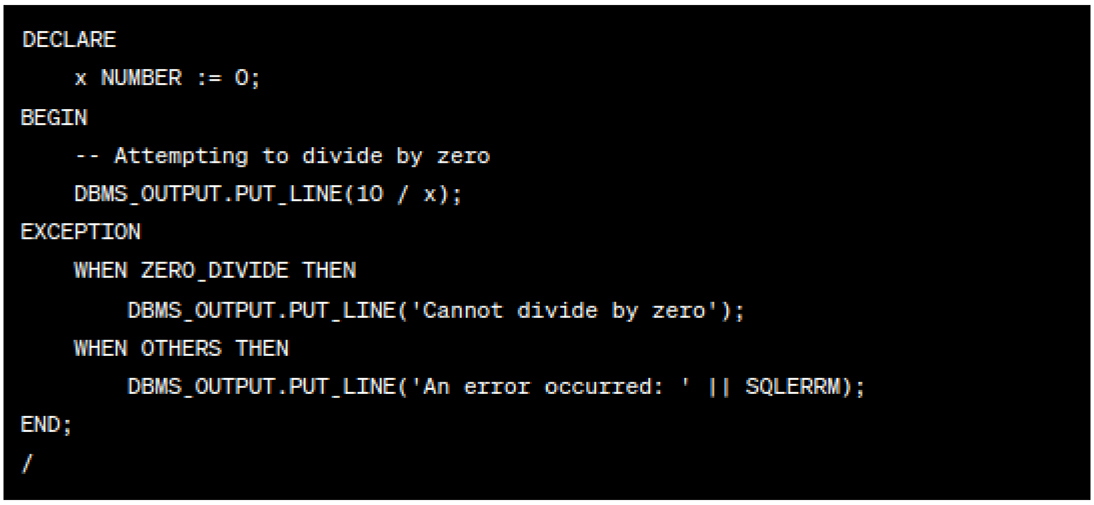
External Exceptions: External exceptions are errors that you define and raise explicitly in your PL/SQL code. You can use the RAISE statement to raise a user-defined exception.

```
DECLARE
    x NUMBER := 10;
    y NUMBER := 20;
BEGIN
    -- Check for a business rule violation
    IF x > y THEN
        RAISE_APPLICATION_ERROR(-20001, 'x should not be greater than y');
    END IF;

    -- Rest of the code
    DBMS_OUTPUT.PUT_LINE('No business rule violation');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

In this example, if the business rule is violated (i.e., if x is greater than y), a user-defined exception with the error code -20001 and a custom error message will be raised. The OTHERS handler will catch any unhandled exception, providing a generic error message.

Handling Specific Exceptions:

You can also handle specific exceptions based on the error code.



```
DECLARE
    x NUMBER := 0;
BEGIN
    -- Attempting to divide by zero
    DBMS_OUTPUT.PUT_LINE(10 / x);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Cannot divide by zero');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

In this example, the ZERO_DIVIDE exception is caught specifically, and a custom error message is printed. The OTHERS handler provides a catch-all for any other unhandled exception.

It's important to handle errors gracefully in PL/SQL to ensure that your application can recover or provide meaningful information to users and developers. Always consider the specific context and business rules when designing error-handling mechanisms.

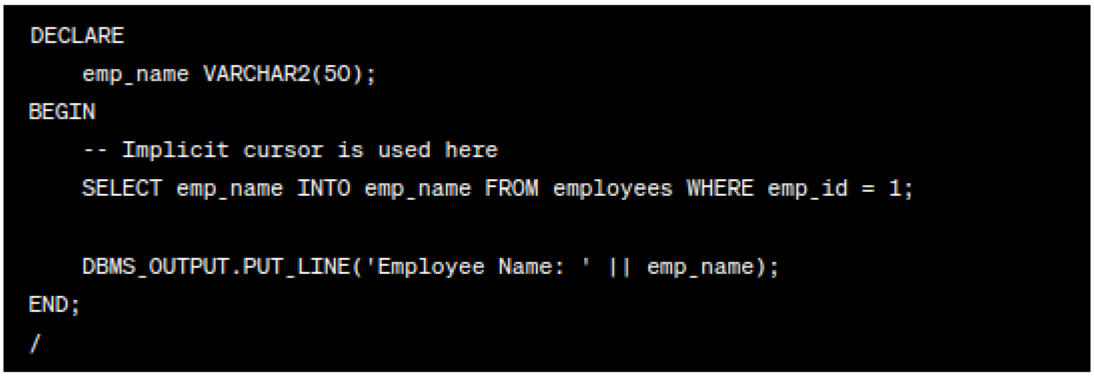
Practical 8

Aim: Using various types of Cursors

In PL/SQL, cursors are used to process the result set of a query. Cursors allow you to iterate over the rows returned by a query and perform operations on each row. There are two main types of cursors: implicit and explicit. Additionally, explicit cursors can be further classified into three types: FOR, WHILE, and SELECT.

1. Implicit Cursor:

Implicit cursors are automatically created by Oracle when a SQL statement is executed in a PL/SQL block. They are used for single-row queries.

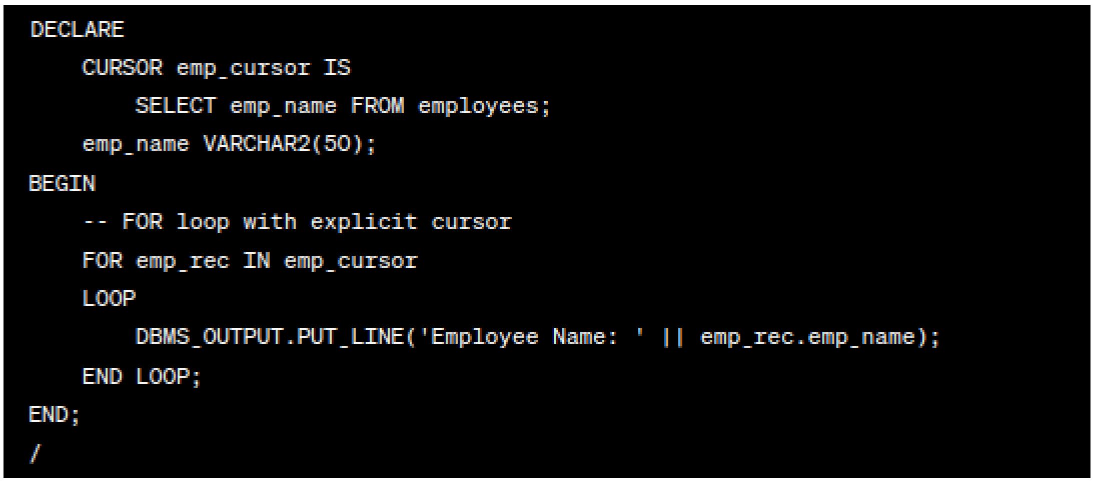


```
DECLARE
    emp_name VARCHAR2(50);
BEGIN
    -- Implicit cursor is used here
    SELECT emp_name INTO emp_name FROM employees WHERE emp_id = 1;

    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_name);
END;
/
```

In this example, an implicit cursor is created automatically for the SELECT statement. The result of the query is fetched into the emp_name variable.

2. Explicit Cursor - FOR Loop:

Explicit cursors are created by the developer and offer more control over the result set. The FOR loop implicitly opens, fetches, and closes the cursor.



```
DECLARE
    CURSOR emp_cursor IS
        SELECT emp_name FROM employees;
    emp_name VARCHAR2(50);
BEGIN
    -- FOR loop with explicit cursor
    FOR emp_rec IN emp_cursor
    LOOP
        DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_rec.emp_name);
    END LOOP;
END;
/
```

In this example, an explicit cursor named emp_cursor is defined, and a FOR loop is used to iterate over the result set and print each employee name.

3. Explicit Cursor - WHILE Loop:

The WHILE loop provides a way to manually control the fetching of rows from the cursor.

```
DECLARE
    CURSOR emp_cursor IS
        SELECT emp_name FROM employees;
    emp_rec emp_cursor%ROWTYPE;
BEGIN
    -- WHILE loop with explicit cursor
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO emp_rec;
        EXIT WHEN emp_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_rec.emp_name);
    END LOOP;
    CLOSE emp_cursor;
END;
/
```

Here, the WHILE loop is used to manually fetch rows from the cursor until there are no more rows (emp_cursor%NOTFOUND).

4. Explicit Cursor - SELECT:

The SELECT cursor allows more manual control over cursor operations.

```
DECLARE
    CURSOR emp_cursor IS
        SELECT emp_name FROM employees;
    emp_rec emp_cursor%ROWTYPE;
BEGIN
    -- Explicit cursor with SELECT statement
    OPEN emp_cursor;
    FETCH emp_cursor INTO emp_rec;
    WHILE emp_cursor%FOUND
    LOOP
        DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_rec.emp_name);
        FETCH emp_cursor INTO emp_rec;
    END LOOP;
    CLOSE emp_cursor;
END;
/
```

Practical 9

Aim: How to run Stored Procedures and Functions

Running stored procedures and functions in PL/SQL involves invoking them from a PL/SQL block or another stored procedure. Here are examples of how to run both stored procedures and functions:

Running Stored Procedure:

Assuming you have a stored procedure named update_employee_salary:

```
CREATE OR REPLACE PROCEDURE update_employee_salary(
    p_emp_id IN NUMBER,
    p_new_salary IN NUMBER
) AS
BEGIN
    -- Your logic to update the employee salary goes here
    UPDATE employees
    SET emp_salary = p_new_salary
    WHERE emp_id = p_emp_id;

    COMMIT; -- Commit the changes
END;
/
```

You can run this stored procedure using an anonymous PL/SQL block:

```
DECLARE
    emp_id_to_update NUMBER := 1;
    new_salary NUMBER := 60000;
BEGIN
    update_employee_salary(emp_id_to_update, new_salary);
END;
/
```

This block declares variables, assigns values, and then calls the update_employee_salary stored procedure.

Running Stored Function:
Assuming you have a stored function named get_employee_salary:

```
CREATE OR REPLACE FUNCTION get_employee_salary(
    p_emp_id IN NUMBER
) RETURN NUMBER AS
    v_salary NUMBER;
BEGIN
    -- Your logic to retrieve the employee salary goes here
    SELECT emp_salary INTO v_salary
    FROM employees
    WHERE emp_id = p_emp_id;

    RETURN v_salary;
END;
/
```

You can run this stored function in a PL/SQL block or SQL query:

```
DECLARE
    emp_id_to_query NUMBER := 1;
    salary NUMBER;
BEGIN
    salary := get_employee_salary(emp_id_to_query);
    DBMS_OUTPUT.PUT_LINE('Employee Salary: ' || salary);
END;
/
```

In this block, the get_employee_salary function is called, and the returned value is stored in the salary variable.

Alternatively, you can run the stored function in a SQL query:

```
SELECT get_employee_salary(1) AS employee_salary FROM DUAL;
```

This query calls the stored function and returns the result.

When running stored procedures or functions, ensure you have the necessary permissions to execute them. Additionally, provide appropriate input parameters and handle any exceptions that may occur during execution.

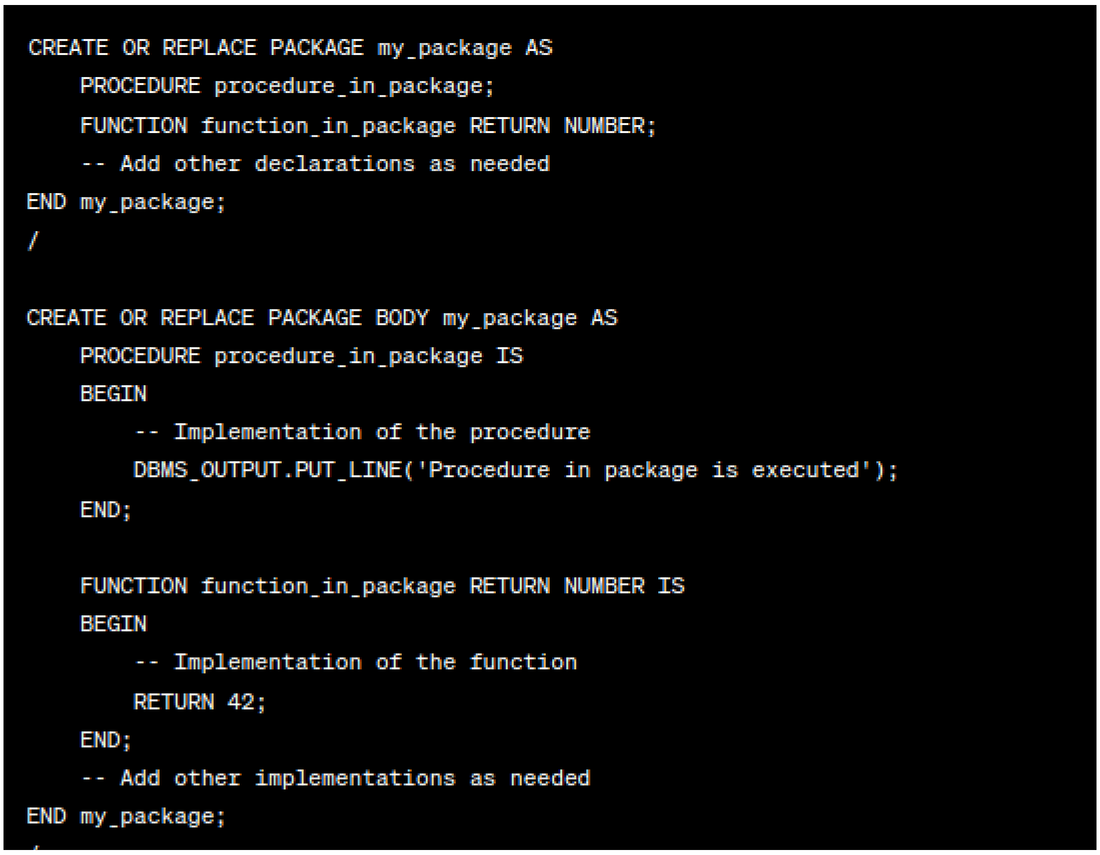
Practical 10

Aim: Creating Packages and applying Triggers

Creating packages and applying triggers are common tasks in PL/SQL for organizing and encapsulating code logic and responding to database events. Here's an overview of creating packages and applying triggers:

Creating Packages:

A package is a container for related procedures, functions, variables, and other PL/SQL constructs. It allows you to organize and modularize your code. Below is an example of creating a simple package:

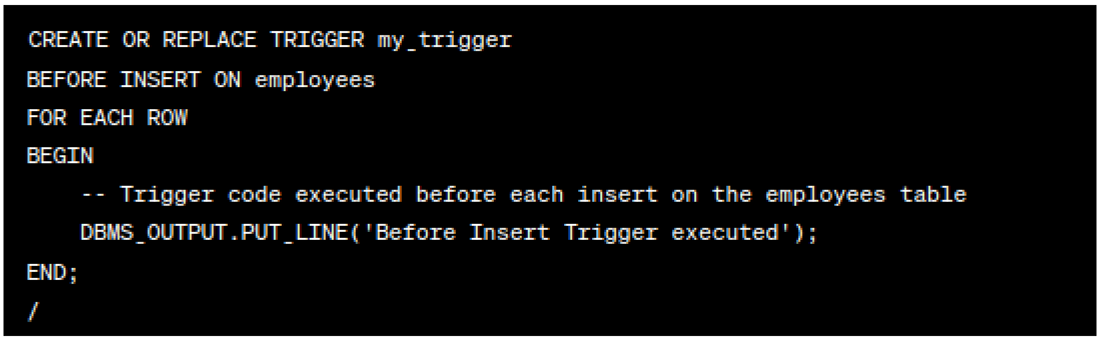


```
CREATE OR REPLACE PACKAGE my_package AS
    PROCEDURE procedure_in_package;
    FUNCTION function_in_package RETURN NUMBER;
    -- Add other declarations as needed
END my_package;
/

CREATE OR REPLACE PACKAGE BODY my_package AS
    PROCEDURE procedure_in_package IS
    BEGIN
        -- Implementation of the procedure
        DBMS_OUTPUT.PUT_LINE('Procedure in package is executed');
    END;

    FUNCTION function_in_package RETURN NUMBER IS
    BEGIN
        -- Implementation of the function
        RETURN 42;
    END;
    -- Add other implementations as needed
END my_package;
```

In this example, my_package is the package specification, and my_package is the package body. You can declare procedures, functions, and variables in the specification and implement them in the body.

Applying Triggers:

Triggers are pieces of PL/SQL code that are automatically executed (or "triggered") in response to events in the database. Here's an example of creating a simple trigger:

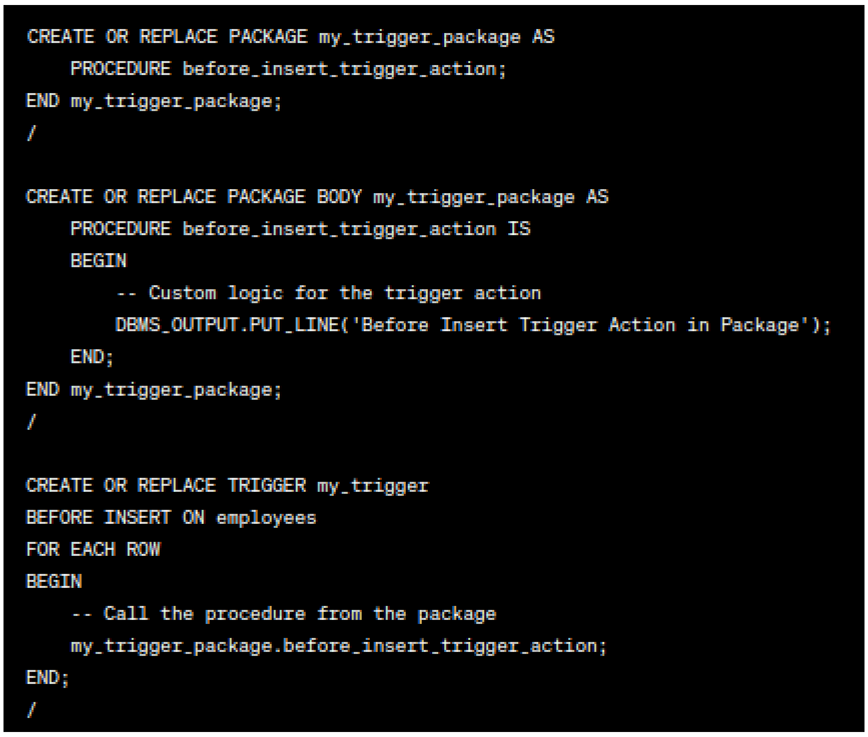


```
CREATE OR REPLACE TRIGGER my_trigger
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    -- Trigger code executed before each insert on the employees table
    DBMS_OUTPUT.PUT_LINE('Before Insert Trigger executed');
END;
/
```

In this example, my_trigger is a trigger that fires before each insert on the employees table. You can define triggers for different events such as BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE, and AFTER DELETE.

Using Triggers with a Package:

You can also use triggers in conjunction with packages. For example:



```
CREATE OR REPLACE PACKAGE my_trigger_package AS
    PROCEDURE before_insert_trigger_action;
END my_trigger_package;
/

CREATE OR REPLACE PACKAGE BODY my_trigger_package AS
    PROCEDURE before_insert_trigger_action IS
    BEGIN
        -- Custom logic for the trigger action
        DBMS_OUTPUT.PUT_LINE('Before Insert Trigger Action in Package');
    END;
END my_trigger_package;
/

CREATE OR REPLACE TRIGGER my_trigger
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    -- Call the procedure from the package
    my_trigger_package.before_insert_trigger_action;
END;
/
```

In this example, the before_insert_trigger_action procedure is defined in the package my_trigger_package, and the trigger my_trigger calls this procedure before each insert on the employees table.

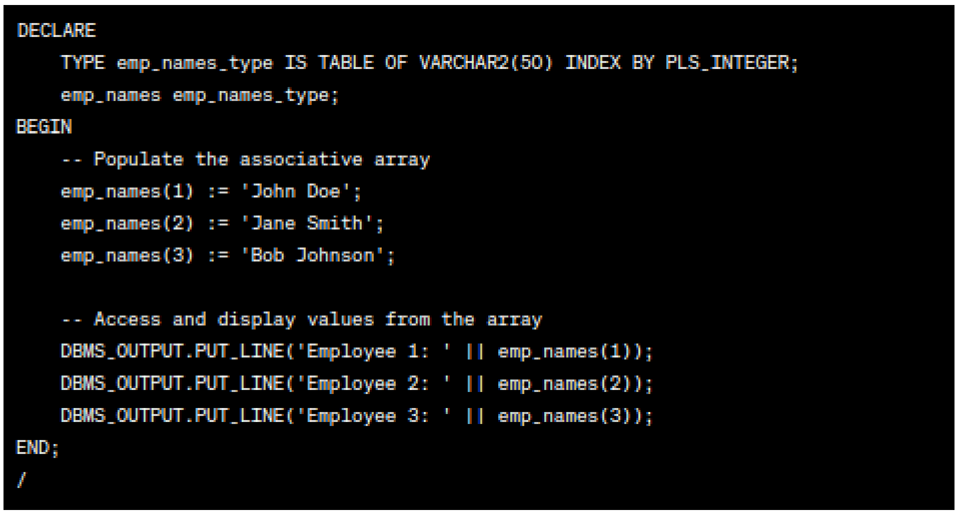
Practical 11

Aim: Creating Arrays and Nested Tables.

In PL/SQL, collections such as arrays and nested tables provide a way to store multiple values in a single variable. Here are examples of creating arrays and nested tables:

1. Associative Arrays (Index-By Tables):

Associative arrays are also known as index-by tables or simply arrays. They are similar to arrays in other programming languages and use an index to access elements.

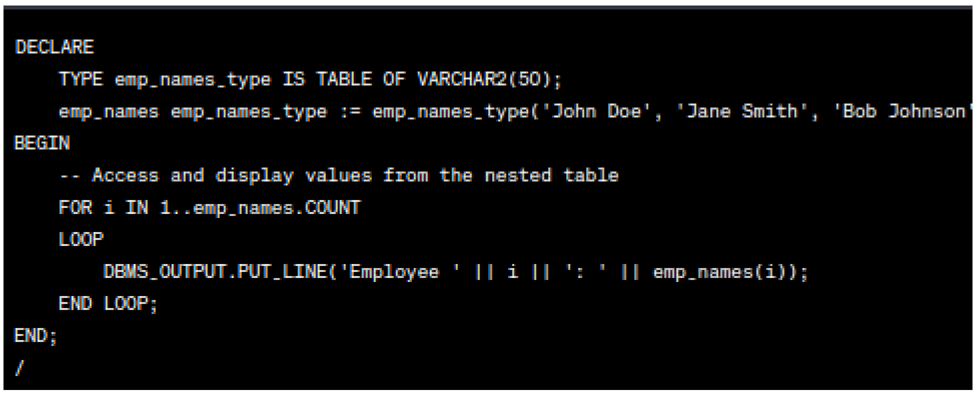


```
DECLARE
    TYPE emp_names_type IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
    emp_names emp_names_type;
BEGIN
    -- Populate the associative array
    emp_names(1) := 'John Doe';
    emp_names(2) := 'Jane Smith';
    emp_names(3) := 'Bob Johnson';

    -- Access and display values from the array
    DBMS_OUTPUT.PUT_LINE('Employee 1: ' || emp_names(1));
    DBMS_OUTPUT.PUT_LINE('Employee 2: ' || emp_names(2));
    DBMS_OUTPUT.PUT_LINE('Employee 3: ' || emp_names(3));
END;
/
```

In this example, an associative array named emp_names is created, and values are assigned to specific indices. The INDEX BY PLS_INTEGER clause indicates that the index is an integer.

2. Nested Tables:

Nested tables are similar to associative arrays but can have their size adjusted dynamically.

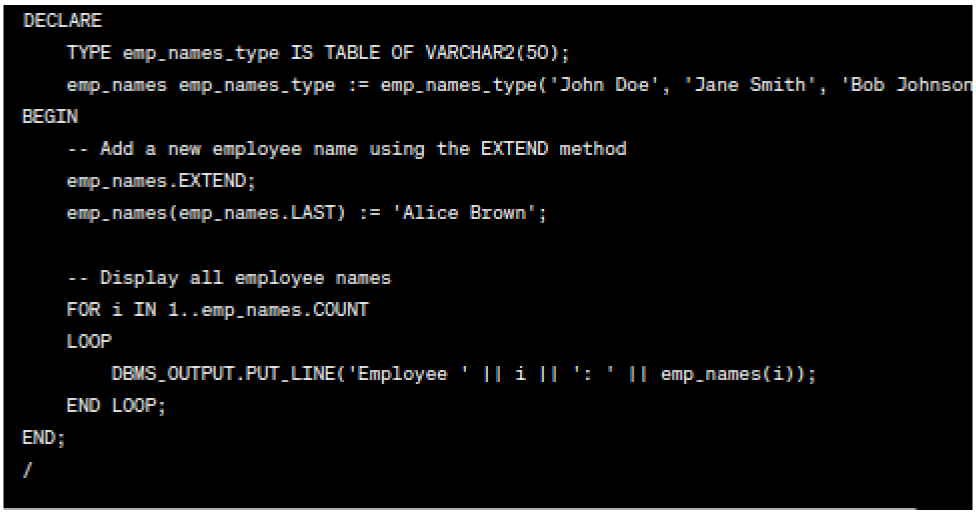


```
DECLARE
    TYPE emp_names_type IS TABLE OF VARCHAR2(50);
    emp_names emp_names_type := emp_names_type('John Doe', 'Jane Smith', 'Bob Johnson'
BEGIN
    -- Access and display values from the nested table
    FOR i IN 1..emp_names.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('Employee ' || i || ': ' || emp_names(i));
    END LOOP;
END;
/
```

In this example, a nested table named emp_names is created and initialized with values. The emp_names.COUNT function is used to determine the number of elements in the nested table.

3. Nested Table Methods:

Nested tables have useful methods for manipulating data. Here's an example using the EXTEND method:



```
DECLARE
    TYPE emp_names_type IS TABLE OF VARCHAR2(50);
    emp_names emp_names_type := emp_names_type('John Doe', 'Jane Smith', 'Bob Johnson
BEGIN
    -- Add a new employee name using the EXTEND method
    emp_names.EXTEND;
    emp_names(emp_names.LAST) := 'Alice Brown';

    -- Display all employee names
    FOR i IN 1..emp_names.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('Employee ' || i || ': ' || emp_names(i));
    END LOOP;
END;
/
```

In this example, the EXTEND method is used to add a new element to the nested table, and the emp_names.LAST function is used to access the last index.