LEARNING PL/SQL

AN ADVANCED LEARNING GUIDE

RANDALL FADLER | AUGUST 2024



Preface

Welcome to the **Advanced Learning Guide to Oracle PL/SQL**. This guide is designed for those who have a solid foundation in Oracle PL/SQL and are looking to deepen their understanding and enhance their skills. Whether you are a developer, database administrator, or IT professional, this guide will provide you with the advanced techniques and best practices needed to master Oracle PL/SQL.

Assumptions

Before diving into the advanced topics, it is assumed that you:

- Are familiar with using an SQLPlus.
- Have a solid understanding of Oracle PL/SQL, including basic syntax, control structures, and standard functions.
- Know how to save and manage PL/SQL files for future use and reference.

Topics Covered

This guide covers a wide range of advanced topics, each designed to build upon your existing knowledge and take your PL/SQL skills to the next level. The following sections will be explored in detail:

1. **Advanced Collections**
    - Learn about nested collections, VARRAYs, and associative arrays, and how to use them to manage complex data structures.
2. **Advanced Bulk Operations**
    - Discover techniques for efficient data processing using bulk collect and bulk bind operations.
3. **Advanced Dynamic SQL**
    - Understand how to construct and execute dynamic SQL statements for flexible and powerful database interactions.
4. **Advanced Error Handling**
    - Explore advanced error handling techniques to create robust and resilient PL/SQL code.
5. **Advanced Cursors**
    - Delve into the use of advanced cursor techniques, including cursor variables and ref cursors.
6. **Advanced Procedures and Functions**
    - Enhance your skills in creating and managing advanced procedures and functions, including overloading and pipelined functions.
7. **Advanced Packages**
    - Learn how to design and implement advanced packages to encapsulate and organize your PL/SQL code effectively.
8. **Advanced Triggers**
    - Gain insights into creating and managing complex triggers for sophisticated database automation.

9. **Performance Tuning**
    - Discover best practices and techniques for optimizing the performance of your PL/SQL code.
10. **Security**
    - Understand fine-grained security
11. **Best Practices**
    - Learn about the best practices for writing clean, maintainable, and efficient PL/SQL code.
12. **Resources for Further Learning**
    - Explore additional resources, including books, online courses, and community forums, to continue your learning journey.

Conclusion

By the end of this guide, you will have a comprehensive understanding of advanced Oracle PL/SQL programming techniques. You will be equipped with the knowledge and skills to tackle complex database challenges and create efficient, high-performance PL/SQL applications.

Let's embark on this journey to master the advanced aspects of Oracle PL/SQL together!

Intermediate PL/SQL Concepts

1. **Nested Collections**

   Nested collections in Oracle PL/SQL are a powerful feature that allows you to create complex data structures. They are particularly useful when you need to store and manipulate sets of data that are related to each other. Let's dive into the details.

   A nested collection is essentially a collection within another collection. Oracle PL/SQL supports three types of collections: associative arrays (also known as index-by tables), nested tables, and VARRAYs. Among these, nested tables and VARRAYs can be nested within each other to create more complex data structures.

   ### Nested Tables

   A nested table is a collection that can store an arbitrary number of elements. Unlike VARRAYs, nested tables do not have a fixed size, and they can be sparse, meaning that they can have gaps between elements.

   Example:

   Let's create a nested table of numbers and then nest it within another table.

   Define the Types:

```
-- Define a type for the inner nested table
CREATE OR REPLACE TYPE number_table AS TABLE OF NUMBER;


-- Define a type for the outer nested table
CREATE OR REPLACE TYPE nested_number_table AS TABLE OF number_table;
```

   Use the Types in PL/SQL:

   SQL

```
DECLARE
  -- Declare a variable of the outer nested table type
  outer_table nested_number_table := nested_number_table();
  -- Declare a variable of the inner nested table type
  inner_table number_table := number_table();
BEGIN
  -- Populate the inner table
  inner_table.EXTEND(3);
  inner_table(1) := 10;
  inner_table(2) := 20;
```

```
  inner_table(3) := 30;


  -- Add the inner table to the outer table
  outer_table.EXTEND;
  outer_table(outer_table.COUNT) := inner_table;


  -- Display the contents of the nested collections
  FOR i IN 1..outer_table.COUNT LOOP
    FOR j IN 1..outer_table(i).COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Element (' || i || ',' || j || '): ' ||
outer_table(i)(j));
    END LOOP;
  END LOOP;
END;
```

In this example, we first define two types: number_table for the inner nested table and nested_number_table for the outer nested table. We then declare variables of these types and populate the inner table with some numbers. Finally, we add the inner table to the outer table and display the contents.

VARRAYs

A VARRAY (variable-size array) is a collection with a fixed maximum size. Unlike nested tables, VARRAYs are always dense, meaning that they do not have gaps between elements.

Example:

Let's create a VARRAY of numbers and then nest it within another VARRAY.

Define the Types:

```
-- Define a type for the inner VARRAY
CREATE OR REPLACE TYPE number_varray AS VARRAY(5) OF NUMBER;


-- Define a type for the outer VARRAY
CREATE OR REPLACE TYPE nested_number_varray AS VARRAY(5) OF number_varray;
```

Use the Types in PL/SQL:

```
DECLARE
  -- Declare a variable of the outer VARRAY type
```

```
  outer_varray nested_number_varray := nested_number_varray();
  -- Declare a variable of the inner VARRAY type
  inner_varray number_varray := number_varray();
BEGIN
  -- Populate the inner VARRAY
  inner_varray.EXTEND(3);
  inner_varray(1) := 10;
  inner_varray(2) := 20;
  inner_varray(3) := 30;

  -- Add the inner VARRAY to the outer VARRAY
  outer_varray.EXTEND;
  outer_varray(outer_varray.COUNT) := inner_varray;

  -- Display the contents of the nested collections
  FOR i IN 1..outer_varray.COUNT LOOP
    FOR j IN 1..outer_varray(i).COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Element (' || i || ',' || j || '): ' ||
outer_varray(i)(j));
    END LOOP;
  END LOOP;
END;
```

In this example, we define two types: number_varray for the inner VARRAY and nested_number_varray for the outer VARRAY. We then declare variables of these types and populate the inner VARRAY with some numbers. Finally, we add the inner VARRAY to the outer VARRAY and display the contents.

Use Cases

Nested collections are particularly useful in scenarios where you need to model complex data relationships. For example, you might use nested collections to represent a hierarchy of data, such as departments within a company, where each department has multiple employees.

Here are some detailed use cases for you to consider.

In this scenario, we will use nested tables to model the hierarchy. We'll create a nested table to represent employees and another nested table to represent departments. Each department will contain a nested table of employees.

Step-by-Step Example

1. **Define the Types:**

First, we need to define the types for employees and departments.

```
-- Define a type for the employee table
CREATE OR REPLACE TYPE employee_type AS OBJECT (
  employee_id NUMBER,
  employee_name VARCHAR2(50)
);

CREATE OR REPLACE TYPE employee_table AS TABLE OF employee_type;

-- Define a type for the department table
CREATE OR REPLACE TYPE department_type AS OBJECT (
  department_id NUMBER,
  department_name VARCHAR2(50),
  employees employee_table
);

CREATE OR REPLACE TYPE department_table AS TABLE OF department_type;
```

2. **Use the Types in PL/SQL:**

Next, we will use these types in a PL/SQL block to create and manipulate the hierarchical data.

```
DECLARE
  -- Declare a variable of the department table type
```

```
  departments department_table := department_table();
  -- Declare variables for individual departments and employees
  dept1 department_type;
  dept2 department_type;
  emp1 employee_type;
  emp2 employee_type;
  emp3 employee_type;
  emp4 employee_type;
BEGIN
  -- Create employees
  emp1 := employee_type(1, 'John Doe');
  emp2 := employee_type(2, 'Jane Smith');
  emp3 := employee_type(3, 'Alice Johnson');
  emp4 := employee_type(4, 'Bob Brown');

  -- Create departments and add employees to them
  dept1 := department_type(101, 'HR', employee_table(emp1, emp2));
  dept2 := department_type(102, 'IT', employee_table(emp3, emp4));

  -- Add departments to the department table
  departments.EXTEND(2);
  departments(1) := dept1;
  departments(2) := dept2;

  -- Display the hierarchical data
  FOR i IN 1..departments.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Department ID: ' || departments(i).department_id);
    DBMS_OUTPUT.PUT_LINE('Department Name: ' ||
departments(i).department_name);
    FOR j IN 1..departments(i).employees.COUNT LOOP
```

```
      DBMS_OUTPUT.PUT_LINE('  Employee ID: ' ||
departments(i).employees(j).employee_id);

      DBMS_OUTPUT.PUT_LINE('  Employee Name: ' ||
departments(i).employees(j).employee_name);

    END LOOP;

  END LOOP;

END;
```

In this example:

- We define an employee_type object to represent individual employees and an employee_table to hold a collection of employees.
- We define a department_type object to represent departments, which includes a nested table of employees.
- We create instances of employees and departments, populate the departments with employees, and then add the departments to the department_table.
- Finally, we use nested loops to display the hierarchical data.

Benefits of Using Nested Collections

1. **Organized Data Structure:** Nested collections allow you to maintain a clear and organized structure, making it easier to manage and manipulate related data.
2. **Flexibility:** You can easily add, remove, or update elements within the nested collections without affecting the overall structure.
3. **Performance:** Nested collections can improve performance by reducing the need for multiple joins and complex queries.

Here are some detailed examples to review.

1. Order Management System

In an order management system, you might need to store orders and their associated items. Each order can have multiple items, and using nested collections can help you manage this relationship effectively.

Step-by-Step Example

Create the types:

Order Type:

```
CREATE OR REPLACE TYPE item_type AS OBJECT (

  item_id NUMBER,

  item_name VARCHAR2(50),

  quantity NUMBER
```

```
);

CREATE OR REPLACE TYPE item_table AS TABLE OF item_type;

CREATE OR REPLACE TYPE order_type AS OBJECT (
  order_id NUMBER,
  customer_name VARCHAR2(50),
  items item_table
);
CREATE OR REPLACE TYPE order_table AS TABLE OF order_type;
```

Usage in PL/SQL:

```
DECLARE
  orders order_table := order_table();
  order1 order_type;
  order2 order_type;
  item1 item_type;
  item2 item_type;
  item3 item_type;
BEGIN
  item1 := item_type(1, 'Laptop', 2);
  item2 := item_type(2, 'Mouse', 5);
  item3 := item_type(3, 'Keyboard', 3);

  order1 := order_type(101, 'Alice', item_table(item1, item2));
  order2 := order_type(102, 'Bob', item_table(item3));

  orders.EXTEND(2);
  orders(1) := order1;
  orders(2) := order2;
```

```
  FOR i IN 1..orders.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Order ID: ' || orders(i).order_id);
    DBMS_OUTPUT.PUT_LINE('Customer Name: ' || orders(i).customer_name);
    FOR j IN 1..orders(i).items.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('  Item ID: ' || orders(i).items(j).item_id);
      DBMS_OUTPUT.PUT_LINE('  Item Name: ' || orders(i).items(j).item_name);
      DBMS_OUTPUT.PUT_LINE('  Quantity: ' || orders(i).items(j).quantity);
    END LOOP;
  END LOOP;
END;
```

2. Student Enrollment System

In a student enrollment system, you might need to store students and their enrolled courses. Each student can enroll in multiple courses, and nested collections can help manage this data.

Example:

Student Type:

```
CREATE OR REPLACE TYPE course_type AS OBJECT (
  course_id NUMBER,
  course_name VARCHAR2(50)
);

CREATE OR REPLACE TYPE course_table AS TABLE OF course_type;

CREATE OR REPLACE TYPE student_type AS OBJECT (
  student_id NUMBER,
  student_name VARCHAR2(50),
  courses course_table
);
CREATE OR REPLACE TYPE student_table AS TABLE OF student_type;
```

Usage in PL/SQL:

```
DECLARE
  students student_table := student_table();
  student1 student_type;
  student2 student_type;
  course1 course_type;
  course2 course_type;
  course3 course_type;
BEGIN
  course1 := course_type(1, 'Math');
  course2 := course_type(2, 'Science');
  course3 := course_type(3, 'History');

  student1 := student_type(101, 'John', course_table(course1, course2));
  student2 := student_type(102, 'Jane', course_table(course3));

  students.EXTEND(2);
  students(1) := student1;
  students(2) := student2;

  FOR i IN 1..students.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Student ID: ' || students(i).student_id);
    DBMS_OUTPUT.PUT_LINE('Student Name: ' || students(i).student_name);
    FOR j IN 1..students(i).courses.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('  Course ID: ' ||
students(i).courses(j).course_id);
      DBMS_OUTPUT.PUT_LINE('  Course Name: ' ||
students(i).courses(j).course_name);
    END LOOP;
  END LOOP;
```

```
END;
```

3. Inventory Management System

In an inventory management system, you might need to store warehouses and their associated products. Each warehouse can store multiple products, and nested collections can help manage this relationship.

Example:

Warehouse Type:

```
CREATE OR REPLACE TYPE product_type AS OBJECT (
  product_id NUMBER,
  product_name VARCHAR2(50),
  stock NUMBER
);

CREATE OR REPLACE TYPE product_table AS TABLE OF product_type;

CREATE OR REPLACE TYPE warehouse_type AS OBJECT (
  warehouse_id NUMBER,
  warehouse_name VARCHAR2(50),
  products product_table
);
CREATE OR REPLACE TYPE warehouse_table AS TABLE OF warehouse_type;
```

- **Usage in PL/SQL:**

```
DECLARE
  warehouses warehouse_table := warehouse_table();
  warehouse1 warehouse_type;
  warehouse2 warehouse_type;
  product1 product_type;
  product2 product_type;
  product3 product_type;
BEGIN
  product1 := product_type(1, 'Widget A', 100);
```

```
  product2 := product_type(2, 'Widget B', 200);
  product3 := product_type(3, 'Widget C', 150);

  warehouse1 := warehouse_type(101, 'Warehouse 1', product_table(product1,
product2));
  warehouse2 := warehouse_type(102, 'Warehouse 2', product_table(product3));

  warehouses.EXTEND(2);
  warehouses(1) := warehouse1;
  warehouses(2) := warehouse2;

  FOR i IN 1..warehouses.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Warehouse ID: ' || warehouses(i).warehouse_id);
    DBMS_OUTPUT.PUT_LINE('Warehouse Name: ' || warehouses(i).warehouse_name);
    FOR j IN 1..warehouses(i).products.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('  Product ID: ' ||
warehouses(i).products(j).product_id);
      DBMS_OUTPUT.PUT_LINE('  Product Name: ' ||
warehouses(i).products(j).product_name);
      DBMS_OUTPUT.PUT_LINE('  Stock: ' || warehouses(i).products(j).stock);
    END LOOP;
  END LOOP;
END;
```

4. Project Management System

In a project management system, you might need to store projects and their associated tasks. Each project can have multiple tasks, and nested collections can help manage this data.

Example:

- **Project Type:**

```
CREATE OR REPLACE TYPE task_type AS OBJECT (
  task_id NUMBER,
  task_name VARCHAR2(50),
```

```
  status VARCHAR2(20)
);

CREATE OR REPLACE TYPE task_table AS TABLE OF task_type;

CREATE OR REPLACE TYPE project_type AS OBJECT (
  project_id NUMBER,
  project_name VARCHAR2(50),
  tasks task_table
);
CREATE OR REPLACE TYPE project_table AS TABLE OF project_type;
```

- **Usage in PL/SQL:**

```
DECLARE
  projects project_table := project_table();
  project1 project_type;
  project2 project_type;
  task1 task_type;
  task2 task_type;
  task3 task_type;
BEGIN
  task1 := task_type(1, 'Design', 'Completed');
  task2 := task_type(2, 'Development', 'In Progress');
  task3 := task_type(3, 'Testing', 'Pending');

  project1 := project_type(101, 'Project Alpha', task_table(task1, task2));
  project2 := project_type(102, 'Project Beta', task_table(task3));

  projects.EXTEND(2);
  projects(1) := project1;
```

```
  projects(2) := project2;


  FOR i IN 1..projects.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Project ID: ' || projects(i).project_id);
    DBMS_OUTPUT.PUT_LINE('Project Name: ' || projects(i).project_name);
    FOR j IN 1..projects(i).tasks.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('  Task ID: ' || projects(i).tasks(j).task_id);
      DBMS_OUTPUT.PUT_LINE('  Task Name: ' ||
projects(i).tasks(j).task_name);
      DBMS_OUTPUT.PUT_LINE('  Status: ' || projects(i).tasks(j).status);
    END LOOP;
  END LOOP;
END;
```

Learn more

1guru99.com2oracle-base.com3tutorialspoint.com4asktom.oracle.com

2. **Advanced Bulk Operations**

Oracle PL/SQL provides advanced bulk operations to handle large volumes of data efficiently. One such feature is bulk binding with the RETURNING clause, which allows you to retrieve values from rows affected by INSERT, UPDATE, or DELETE statements in bulk. This can significantly improve performance by reducing context switches between the SQL and PL/SQL engines.

Key Concepts:

- Bulk Binding: Using FORALL to execute DML statements in bulk.
- RETURNING Clause: Retrieves values from affected rows and stores them in collections.
- Example of Bulk Binding with RETURNING Clause:

Let's consider an example where we update the salaries of employees and retrieve the updated salaries using the RETURNING clause.

Step 1: Create a Table and Insert Sample Data

```
CREATE TABLE employees (
  employee_id NUMBER PRIMARY KEY,
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),
```

```
  salary NUMBER
);


INSERT INTO employees (employee_id, first_name, last_name, salary) VALUES (1,
'John', 'Doe', 50000);

INSERT INTO employees (employee_id, first_name, last_name, salary) VALUES (2,
'Jane', 'Smith', 60000);

INSERT INTO employees (employee_id, first_name, last_name, salary) VALUES (3,
'Alice', 'Johnson', 70000);

COMMIT;
```

Step 2: Bulk Update Salaries and Retrieve Updated Values

```
DECLARE
  TYPE employee_id_table IS TABLE OF employees.employee_id%TYPE;
  TYPE salary_table IS TABLE OF employees.salary%TYPE;

  l_employee_ids employee_id_table := employee_id_table(1, 2, 3);
  l_new_salaries salary_table := salary_table(55000, 65000, 75000);
  l_updated_salaries salary_table;
BEGIN
  -- Bulk update salaries and retrieve updated values
  FORALL i IN 1..l_employee_ids.COUNT
    UPDATE employees
    SET salary = l_new_salaries(i)
    WHERE employee_id = l_employee_ids(i)
    RETURNING salary BULK COLLECT INTO l_updated_salaries;

  -- Display updated salaries
  FOR i IN 1..l_updated_salaries.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || l_updated_salaries(i));
  END LOOP;
```

```
END;
```

Explanation:

Declare Collections: Define collections to hold employee IDs, new salaries, and updated salaries.

Initialize Collections: Initialize the collections with sample data.

Bulk Update with RETURNING Clause: Use FORALL to perform bulk updates on the employees table. The RETURNING clause retrieves the updated salaries and stores them in the l_updated_salaries collection.

Display Updated Salaries: Loop through the l_updated_salaries collection and display the updated salaries.

Key Points to Remember:

- Bulk Binding: Use FORALL to execute DML statements in bulk, reducing context switches.
- RETURNING Clause: Retrieve values from affected rows and store them in collections.
- Performance Improvement: Bulk operations with the RETURNING clause can significantly improve performance when handling large volumes of data.

Oracle PL/SQL Bulk Collect with LIMIT Clause

The BULK COLLECT clause in Oracle PL/SQL allows you to fetch multiple rows from a query into a collection in a single context switch, improving performance. The LIMIT clause can be used with BULK COLLECT to fetch a specific number of rows at a time, which is useful for handling large data sets in manageable chunks.

Key Concepts:

1. **BULK COLLECT**: Fetches multiple rows into a collection in a single operation.
2. **LIMIT Clause**: Specifies the maximum number of rows to fetch at a time.

Example of Using BULK COLLECT with LIMIT Clause:

Let's consider an example where we fetch employee data in chunks of 100 rows at a time.

Step 1: Create a Table and Insert Sample Data

```
CREATE TABLE employees (
  employee_id NUMBER PRIMARY KEY,
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),
  salary NUMBER
);
```

```
-- Insert sample data
BEGIN
  FOR i IN 1..1000 LOOP
    INSERT INTO employees (employee_id, first_name, last_name, salary)
    VALUES (i, 'First' || i, 'Last' || i, 50000 + i);
  END LOOP;
  COMMIT;
END;
```

Step 2: Fetch Data Using BULK COLLECT with LIMIT Clause

```
DECLARE
  TYPE employee_id_table IS TABLE OF employees.employee_id%TYPE;
  TYPE first_name_table IS TABLE OF employees.first_name%TYPE;
  TYPE last_name_table IS TABLE OF employees.last_name%TYPE;
  TYPE salary_table IS TABLE OF employees.salary%TYPE;

  l_employee_ids employee_id_table;
  l_first_names first_name_table;
  l_last_names last_name_table;
  l_salaries salary_table;

  CURSOR c_employees IS
    SELECT employee_id, first_name, last_name, salary
    FROM employees;
BEGIN
  OPEN c_employees;
  LOOP
    FETCH c_employees
    BULK COLLECT INTO l_employee_ids, l_first_names, l_last_names, l_salaries
    LIMIT 100; -- Fetch 100 rows at a time
```

```
    EXIT WHEN l_employee_ids.COUNT = 0;


    -- Process the fetched data
    FOR i IN 1..l_employee_ids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Employee ID: ' || l_employee_ids(i) || ', Name: '
|| l_first_names(i) || ' ' || l_last_names(i) || ', Salary: ' ||
l_salaries(i));
    END LOOP;
  END LOOP;
  CLOSE c_employees;
END;
```

Explanation:

1. **Declare Collections**: Define collections to hold employee IDs, first names, last names, and salaries.
2. **Cursor Definition**: Define a cursor c_employees to select employee data.
3. **Open Cursor**: Open the cursor to start fetching data.
4. **Fetch Data in Chunks**: Use BULK COLLECT with the LIMIT clause to fetch 100 rows at a time into the collections.
5. **Process Fetched Data**: Loop through the fetched data and process it (e.g., display it using DBMS_OUTPUT.PUT_LINE).
6. **Exit Condition**: Exit the loop when no more rows are fetched.
7. **Close Cursor**: Close the cursor to release resources.

Key Points to Remember:

- **BULK COLLECT**: Fetches multiple rows into collections in a single operation, improving performance.
- **LIMIT Clause**: Fetches a specific number of rows at a time, useful for handling large data sets in manageable chunks.
- **Cursor**: Use a cursor to fetch data in chunks using BULK COLLECT with the LIMIT clause.
- **Processing Data**: Process the fetched data within the loop before fetching the next chunk.

Managing memory with bulk operations

Managing memory effectively is crucial when performing bulk operations in Oracle PL/SQL, especially when dealing with large volumes of data. Bulk operations, such as BULK COLLECT and FORALL, can significantly improve performance by reducing context switches between the SQL and PL/SQL engines. However, they can also consume a lot of memory if not managed properly.

Key Techniques for Managing Memory with Bulk Operations:

1. **Use the LIMIT Clause**: Fetch data in manageable chunks to avoid consuming too much memory at once.
2. **Free Temporary LOBs**: If working with LOBs (Large Objects), ensure you free temporary LOBs to release memory.
3. **Monitor Memory Usage**: Use PL/SQL functions to monitor memory usage and adjust your operations accordingly.

Example of Using the LIMIT Clause:

Let's consider an example where we fetch employee data in chunks of 100 rows at a time to manage memory usage effectively.

Step 1: Create a Table and Insert Sample Data

```
CREATE TABLE employees (
  employee_id NUMBER PRIMARY KEY,
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),
  salary NUMBER
);

-- Insert sample data
BEGIN
  FOR i IN 1..1000 LOOP
    INSERT INTO employees (employee_id, first_name, last_name, salary)
    VALUES (i, 'First' || i, 'Last' || i, 50000 + i);
  END LOOP;
  COMMIT;
END;
```

Step 2: Fetch Data Using BULK COLLECT with LIMIT Clause

```
DECLARE
  TYPE employee_id_table IS TABLE OF employees.employee_id%TYPE;
  TYPE first_name_table IS TABLE OF employees.first_name%TYPE;
  TYPE last_name_table IS TABLE OF employees.last_name%TYPE;
  TYPE salary_table IS TABLE OF employees.salary%TYPE;

  l_employee_ids employee_id_table;
  l_first_names first_name_table;
  l_last_names last_name_table;
  l_salaries salary_table;

  CURSOR c_employees IS
    SELECT employee_id, first_name, last_name, salary
    FROM employees;
BEGIN
  OPEN c_employees;
  LOOP
    FETCH c_employees
    BULK COLLECT INTO l_employee_ids, l_first_names, l_last_names, l_salaries
    LIMIT 100; -- Fetch 100 rows at a time

    EXIT WHEN l_employee_ids.COUNT = 0;

    -- Process the fetched data
    FOR i IN 1..l_employee_ids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Employee ID: ' || l_employee_ids(i) || ', Name: '
|| l_first_names(i) || ' ' || l_last_names(i) || ', Salary: ' ||
l_salaries(i));
    END LOOP;
```

```
  END LOOP;

  CLOSE c_employees;

END;
```

Explanation:

1. **Declare Collections**: Define collections to hold employee IDs, first names, last names, and salaries.
2. **Cursor Definition**: Define a cursor c_employees to select employee data.
3. **Open Cursor**: Open the cursor to start fetching data.
4. **Fetch Data in Chunks**: Use BULK COLLECT with the LIMIT clause to fetch 100 rows at a time into the collections.
5. **Process Fetched Data**: Loop through the fetched data and process it (e.g., display it using DBMS_OUTPUT.PUT_LINE).
6. **Exit Condition**: Exit the loop when no more rows are fetched.
7. **Close Cursor**: Close the cursor to release resources.

Example of Freeing Temporary LOBs:

If you are working with LOBs, it's important to free temporary LOBs to release memory.

```
DECLARE

  l_clob CLOB;

BEGIN

  -- Create a temporary CLOB

  DBMS_LOB.CREATETEMPORARY(l_clob, TRUE);


  -- Write data to the CLOB

  DBMS_LOB.WRITE(l_clob, LENGTH('This is a large document content.'), 1,
'This is a large document content.');


  -- Process the CLOB data

  DBMS_OUTPUT.PUT_LINE('CLOB Length: ' || DBMS_LOB.GETLENGTH(l_clob));


  -- Free the temporary CLOB to release memory

  DBMS_LOB.FREETEMPORARY(l_clob);

END;
```

Key Points to Remember:

- **Use the LIMIT Clause**: Fetch data in manageable chunks to avoid consuming too much memory at once.
- **Free Temporary LOBs**: Ensure you free temporary LOBs to release memory.
- **Monitor Memory Usage**: Use PL/SQL functions to monitor memory usage and adjust your operations accordingly.

3. Advanced Dynamic SQL

Dynamic SQL in Oracle PL/SQL allows you to construct and execute SQL statements dynamically at runtime. This is particularly useful when the exact SQL statement to be executed is not known until runtime, such as when building queries based on user input or other runtime conditions.

Methods for Executing Dynamic SQL:

1. **EXECUTE IMMEDIATE**: Used for single SQL statements.
2. **DBMS_SQL Package**: Provides more control and is used for complex dynamic SQL operations.

Using EXECUTE IMMEDIATE:

EXECUTE IMMEDIATE is a simple and efficient way to execute dynamic SQL statements. It can be used for SELECT, INSERT, UPDATE, DELETE, and even PL/SQL blocks.

Example of EXECUTE IMMEDIATE:

```
DECLARE
  v_sql VARCHAR2(200);
  v_employee_id NUMBER := 1001;
  v_first_name VARCHAR2(50);
BEGIN
  -- Construct the SQL statement
  v_sql := 'SELECT first_name FROM employees WHERE employee_id = :1';

  -- Execute the SQL statement
  EXECUTE IMMEDIATE v_sql INTO v_first_name USING v_employee_id;

  DBMS_OUTPUT.PUT_LINE('First Name: ' || v_first_name);
END;
```

Using DBMS_SQL Package:

The DBMS_SQL package provides more control over dynamic SQL execution and is useful for complex operations, such as handling multiple rows or dynamically constructing PL/SQL blocks.

Example of DBMS_SQL Package:

```
DECLARE
  v_cursor_id NUMBER;
  v_sql VARCHAR2(200);
  v_employee_id NUMBER := 1001;
  v_first_name VARCHAR2(50);
BEGIN
  -- Construct the SQL statement
  v_sql := 'SELECT first_name FROM employees WHERE employee_id = :1';

  -- Open a cursor
  v_cursor_id := DBMS_SQL.OPEN_CURSOR;

  -- Parse the SQL statement
  DBMS_SQL.PARSE(v_cursor_id, v_sql, DBMS_SQL.NATIVE);

  -- Bind the variable
  DBMS_SQL.BIND_VARIABLE(v_cursor_id, ':1', v_employee_id);

  -- Define the column
  DBMS_SQL.DEFINE_COLUMN(v_cursor_id, 1, v_first_name);

  -- Execute the SQL statement
  DBMS_SQL.EXECUTE(v_cursor_id);

  -- Fetch the result
  IF DBMS_SQL.FETCH_ROWS(v_cursor_id) > 0 THEN
```

```
      DBMS_SQL.COLUMN_VALUE(v_cursor_id, 1, v_first_name);
      DBMS_OUTPUT.PUT_LINE('First Name: ' || v_first_name);
    END IF;

    -- Close the cursor
    DBMS_SQL.CLOSE_CURSOR(v_cursor_id);
  END;
```

Dynamic PL/SQL Blocks:

You can also execute dynamic PL/SQL blocks using EXECUTE IMMEDIATE or the DBMS_SQL package.

Example of Dynamic PL/SQL Block with EXECUTE IMMEDIATE:

```
  DECLARE
    v_plsql_block VARCHAR2(500);
  BEGIN
    -- Construct the PL/SQL block
    v_plsql_block := 'BEGIN ' ||
                     '  DBMS_OUTPUT.PUT_LINE(''Hello, World!''); ' ||
                     'END;';

    -- Execute the PL/SQL block
    EXECUTE IMMEDIATE v_plsql_block;
  END;
```

Example of Dynamic PL/SQL Block with DBMS_SQL Package:

```
  DECLARE
    v_cursor_id NUMBER;
    v_plsql_block VARCHAR2(500);
  BEGIN
    -- Construct the PL/SQL block
    v_plsql_block := 'BEGIN ' ||
                     '  DBMS_OUTPUT.PUT_LINE(''Hello, World!''); ' ||
```

```
                    'END;';

        -- Open a cursor
        v_cursor_id := DBMS_SQL.OPEN_CURSOR;

        -- Parse the PL/SQL block
        DBMS_SQL.PARSE(v_cursor_id, v_plsql_block, DBMS_SQL.NATIVE);

        -- Execute the PL/SQL block
        DBMS_SQL.EXECUTE(v_cursor_id);

        -- Close the cursor
        DBMS_SQL.CLOSE_CURSOR(v_cursor_id);
    END;
```

Let's dive into a more advanced example of using dynamic SQL with both EXECUTE IMMEDIATE and the DBMS_SQL package. This example will demonstrate how to dynamically create a table, insert data, update data, and retrieve data using both methods. We'll also include error handling and logging to make it more comprehensive.

Advanced Example: Dynamic SQL with EXECUTE IMMEDIATE and DBMS_SQL

Step 1: Create a Logging Table

First, let's create a table to log the executed SQL statements and any errors that occur.

```
CREATE TABLE dynamic_sql_log (
  log_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  executed_sql CLOB,
  log_message VARCHAR2(4000),
  log_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create a Procedure to Execute Dynamic SQL

We'll create a procedure that uses both EXECUTE IMMEDIATE and the DBMS_SQL package to dynamically create a table, insert data, update data, and retrieve data. The procedure will also log the executed SQL statements and any errors.

```
CREATE OR REPLACE PROCEDURE execute_dynamic_sql IS
  v_sql CLOB;
  v_cursor_id NUMBER;
  v_employee_id NUMBER;
  v_first_name VARCHAR2(50);
  v_last_name VARCHAR2(50);
  v_salary NUMBER;
BEGIN
  -- Step 2.1: Dynamically create a table using EXECUTE IMMEDIATE
  v_sql := 'CREATE TABLE dynamic_employees (' ||
           'employee_id NUMBER PRIMARY KEY, ' ||
           'first_name VARCHAR2(50), ' ||
           'last_name VARCHAR2(50), ' ||
           'salary NUMBER)';
  EXECUTE IMMEDIATE v_sql;
  INSERT INTO dynamic_sql_log (executed_sql, log_message) VALUES (v_sql,
'Table created successfully');

  -- Step 2.2: Dynamically insert data using EXECUTE IMMEDIATE
  v_sql := 'INSERT INTO dynamic_employees (employee_id, first_name,
last_name, salary) VALUES (1, ''John'', ''Doe'', 50000)';
  EXECUTE IMMEDIATE v_sql;
  INSERT INTO dynamic_sql_log (executed_sql, log_message) VALUES (v_sql,
'Data inserted successfully');

  -- Step 2.3: Dynamically update data using EXECUTE IMMEDIATE
  v_sql := 'UPDATE dynamic_employees SET salary = 55000 WHERE employee_id =
1';
  EXECUTE IMMEDIATE v_sql;
  INSERT INTO dynamic_sql_log (executed_sql, log_message) VALUES (v_sql,
'Data updated successfully');
```

```
  -- Step 2.4: Dynamically retrieve data using DBMS_SQL

  v_sql := 'SELECT employee_id, first_name, last_name, salary FROM
dynamic_employees WHERE employee_id = 1';

  v_cursor_id := DBMS_SQL.OPEN_CURSOR;

  DBMS_SQL.PARSE(v_cursor_id, v_sql, DBMS_SQL.NATIVE);

  DBMS_SQL.DEFINE_COLUMN(v_cursor_id, 1, v_employee_id);

  DBMS_SQL.DEFINE_COLUMN(v_cursor_id, 2, v_first_name);

  DBMS_SQL.DEFINE_COLUMN(v_cursor_id, 3, v_last_name);

  DBMS_SQL.DEFINE_COLUMN(v_cursor_id, 4, v_salary);

  DBMS_SQL.EXECUTE(v_cursor_id);


  IF DBMS_SQL.FETCH_ROWS(v_cursor_id) > 0 THEN

    DBMS_SQL.COLUMN_VALUE(v_cursor_id, 1, v_employee_id);

    DBMS_SQL.COLUMN_VALUE(v_cursor_id, 2, v_first_name);

    DBMS_SQL.COLUMN_VALUE(v_cursor_id, 3, v_last_name);

    DBMS_SQL.COLUMN_VALUE(v_cursor_id, 4, v_salary);

    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id || ', Name: ' ||
v_first_name || ' ' || v_last_name || ', Salary: ' || v_salary);

  END IF;


  DBMS_SQL.CLOSE_CURSOR(v_cursor_id);

  INSERT INTO dynamic_sql_log (executed_sql, log_message) VALUES (v_sql,
'Data retrieved successfully');


EXCEPTION

  WHEN OTHERS THEN

    INSERT INTO dynamic_sql_log (executed_sql, log_message) VALUES (v_sql,
'Error: ' || SQLERRM);

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

END;
```

Explanation:

1. **Create Logging Table**: A table dynamic_sql_log is created to log executed SQL statements and any errors.
2. **Procedure Definition**: The execute_dynamic_sql procedure is defined to perform dynamic SQL operations.
3. **Create Table**: A table dynamic_employees is dynamically created using EXECUTE IMMEDIATE.
4. **Insert Data**: Data is dynamically inserted into the dynamic_employees table using EXECUTE IMMEDIATE.
5. **Update Data**: Data is dynamically updated in the dynamic_employees table using EXECUTE IMMEDIATE.
6. **Retrieve Data**: Data is dynamically retrieved from the dynamic_employees table using the DBMS_SQL package.
7. **Logging**: Executed SQL statements and any errors are logged in the dynamic_sql_log table.
8. **Error Handling**: Any errors that occur during the execution are caught and logged.

Key Points to Remember:

- **EXECUTE IMMEDIATE**: Used for simple and efficient execution of dynamic SQL statements.
- **DBMS_SQL Package**: Provides more control for complex dynamic SQL operations.
- **Dynamic PL/SQL Blocks**: Can be executed using both EXECUTE IMMEDIATE and the DBMS_SQL package.
- **Logging and Error Handling**: Log executed SQL statements and handle errors gracefully.

Key Points to Remember:

- **EXECUTE IMMEDIATE**: Simple and efficient for executing single SQL statements and PL/SQL blocks.
- **DBMS_SQL Package**: Provides more control for complex dynamic SQL operations.
- **Dynamic PL/SQL Blocks**: Can be executed using both EXECUTE IMMEDIATE and the DBMS_SQL package.
- **Security Considerations**: Always validate and sanitize inputs to avoid SQL injection vulnerabilities.

- **DBMS_SQL Enhancements**
- Advanced usage of the DBMS_SQL package

Advanced Oracle SQL Statements and Their Use in PL/SQL

Oracle SQL provides a range of advanced features that can be leveraged within PL/SQL to create powerful and efficient database applications. These advanced SQL statements include analytical functions, hierarchical queries, model clauses, and more. Let's explore some of these advanced SQL features and how they can be used within PL/SQL with examples.

1. Analytical Functions

Analytical functions perform calculations across a set of table rows related to the current row. They are useful for complex data analysis and reporting.

Example: Using Analytical Functions in PL/SQL

```
DECLARE
  CURSOR c_analytics IS
    SELECT employee_id, first_name, last_name, salary,
           RANK() OVER (ORDER BY salary DESC) AS salary_rank
    FROM employees;

  v_employee_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_last_name employees.last_name%TYPE;
  v_salary employees.salary%TYPE;
  v_salary_rank NUMBER;
BEGIN
  OPEN c_analytics;
  LOOP
    FETCH c_analytics INTO v_employee_id, v_first_name, v_last_name,
v_salary, v_salary_rank;
    EXIT WHEN c_analytics%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id || ', Name: ' ||
v_first_name || ' ' || v_last_name || ', Salary: ' || v_salary || ', Rank: '
|| v_salary_rank);
  END LOOP;
  CLOSE c_analytics;
END;
```

2. Hierarchical Queries

Hierarchical queries allow you to retrieve data based on a hierarchical relationship, such as an organizational chart.

Example: Using Hierarchical Queries in PL/SQL

```
DECLARE
  CURSOR c_hierarchy IS
    SELECT employee_id, first_name, last_name, manager_id, LEVEL
    FROM employees
    START WITH manager_id IS NULL
    CONNECT BY PRIOR employee_id = manager_id;

  v_employee_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_last_name employees.last_name%TYPE;
  v_manager_id employees.manager_id%TYPE;
  v_level NUMBER;
BEGIN
  OPEN c_hierarchy;
  LOOP
    FETCH c_hierarchy INTO v_employee_id, v_first_name, v_last_name,
v_manager_id, v_level;
    EXIT WHEN c_hierarchy%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id || ', Name: ' ||
v_first_name || ' ' || v_last_name || ', Manager ID: ' || v_manager_id || ',
Level: ' || v_level);
  END LOOP;
  CLOSE c_hierarchy;
END;
```

3. Model Clause

The model clause allows you to perform complex calculations and data manipulation using a spreadsheet-like approach.

Example: Using Model Clause in PL/SQL

```
DECLARE
  CURSOR c_model IS
    SELECT department_id, employee_id, salary
    FROM employees
    MODEL
      PARTITION BY (department_id)
      DIMENSION BY (employee_id)
      MEASURES (salary)
      RULES (
        salary[ANY] = salary[CV()] * 1.1
      );

  v_department_id employees.department_id%TYPE;
  v_employee_id employees.employee_id%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  OPEN c_model;
  LOOP
    FETCH c_model INTO v_department_id, v_employee_id, v_salary;
    EXIT WHEN c_model%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Department ID: ' || v_department_id || ', Employee
ID: ' || v_employee_id || ', New Salary: ' || v_salary);
  END LOOP;
  CLOSE c_model;
END;
```

4. Dynamic SQL

Dynamic SQL allows you to construct and execute SQL statements dynamically at runtime. This is useful when the exact SQL statement to be executed is not known until runtime.

Example: Using Dynamic SQL in PL/SQL

```
DECLARE
  v_sql VARCHAR2(200);
  v_employee_id NUMBER := 1001;
  v_first_name VARCHAR2(50);
BEGIN
  -- Construct the SQL statement
  v_sql := 'SELECT first_name FROM employees WHERE employee_id = :1';

  -- Execute the SQL statement
  EXECUTE IMMEDIATE v_sql INTO v_first_name USING v_employee_id;

  DBMS_OUTPUT.PUT_LINE('First Name: ' || v_first_name);
END;
```

5. Advanced Joins

Advanced joins, such as outer joins, self-joins, and cross joins, allow you to retrieve data from multiple tables based on complex relationships.

Example: Using Advanced Joins in PL/SQL

```
DECLARE
  CURSOR c_joins IS
    SELECT e.employee_id, e.first_name, e.last_name, d.department_name
    FROM employees e
    LEFT JOIN departments d ON e.department_id = d.department_id;

  v_employee_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_last_name employees.last_name%TYPE;
```

```
  v_department_name departments.department_name%TYPE;
BEGIN
  OPEN c_joins;
  LOOP
    FETCH c_joins INTO v_employee_id, v_first_name, v_last_name,
v_department_name;
    EXIT WHEN c_joins%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id || ', Name: ' ||
v_first_name || ' ' || v_last_name || ', Department: ' || v_department_name);
  END LOOP;
  CLOSE c_joins;
END;
```

Key Points to Remember:

- Analytical Functions: Perform calculations across a set of table rows related to the current row.
- Hierarchical Queries: Retrieve data based on hierarchical relationships.
- Model Clause: Perform complex calculations and data manipulation using a spreadsheet-like approach.
- Dynamic SQL: Construct and execute SQL statements dynamically at runtime.
- Advanced Joins: Retrieve data from multiple tables based on complex relationships.

Wrap Up Example

Let's create a simple but inclusive Oracle PL/SQL application for a hospital patient and medication tracking system. This application will include tables with relationships, triggers, packages, functions, and procedures.

Step 1: Create Tables and Relationships

Table: Patients

```
CREATE TABLE patients (
  patient_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),
  date_of_birth DATE,
  gender VARCHAR2(10),
  contact_number VARCHAR2(15)
);
Table: Medications
CREATE TABLE medications (
  medication_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  medication_name VARCHAR2(100),
  dosage VARCHAR2(50),
  side_effects VARCHAR2(200)
);
```

Table: Prescriptions

```
CREATE TABLE prescriptions (
  prescription_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  patient_id NUMBER,
  medication_id NUMBER,
  prescribed_date DATE,
  dosage VARCHAR2(50),
  frequency VARCHAR2(50),
  FOREIGN KEY (patient_id) REFERENCES patients(patient_id),
  FOREIGN KEY (medication_id) REFERENCES medications(medication_id)
```

```
);
```

Step 2: Create Triggers for last Modified Date

Trigger: Update Last Modified Date on Patients Table

```
CREATE OR REPLACE TRIGGER trg_update_last_modified
BEFORE UPDATE ON patients
FOR EACH ROW
BEGIN
  :NEW.last_modified := SYSDATE;
END;
```

Step 3: Create Packages

Package Specification: Patient_Pkg

```
CREATE OR REPLACE PACKAGE patient_pkg IS
  PROCEDURE add_patient(
    p_first_name IN VARCHAR2,
    p_last_name IN VARCHAR2,
    p_date_of_birth IN DATE,
    p_gender IN VARCHAR2,
    p_contact_number IN VARCHAR2
  );
  PROCEDURE update_patient_contact(
    p_patient_id IN NUMBER,
    p_contact_number IN VARCHAR2
  );

  FUNCTION get_patient_details(
    p_patient_id IN NUMBER
  ) RETURN VARCHAR2;
END patient_pkg;
```

Package Body: Patient_Pkg

```
CREATE OR REPLACE PACKAGE BODY patient_pkg IS

  PROCEDURE add_patient(

    p_first_name IN VARCHAR2,

    p_last_name IN VARCHAR2,

    p_date_of_birth IN DATE,

    p_gender IN VARCHAR2,

    p_contact_number IN VARCHAR2

  ) IS

  BEGIN

    INSERT INTO patients (first_name, last_name, date_of_birth, gender,
contact_number)

    VALUES (p_first_name, p_last_name, p_date_of_birth, p_gender,
p_contact_number);

  END add_patient;


  PROCEDURE update_patient_contact(

    p_patient_id IN NUMBER,

    p_contact_number IN VARCHAR2

  ) IS

  BEGIN

    UPDATE patients

    SET contact_number = p_contact_number

    WHERE patient_id = p_patient_id;

  END update_patient_contact;


  FUNCTION get_patient_details(

    p_patient_id IN NUMBER

  ) RETURN VARCHAR2 IS

    v_patient_details VARCHAR2(4000);
```

```
  BEGIN

    SELECT 'Name: ' || first_name || ' ' || last_name || ', DOB: ' ||
TO_CHAR(date_of_birth, 'YYYY-MM-DD') || ', Gender: ' || gender || ', Contact:
' || contact_number

    INTO v_patient_details

    FROM patients

    WHERE patient_id = p_patient_id;


    RETURN v_patient_details;

  END get_patient_details;

END patient_pkg;
```

Step 4: Create Procedures and Functions

Procedure: Add Prescription

```
CREATE OR REPLACE PROCEDURE add_prescription(

  p_patient_id IN NUMBER,

  p_medication_id IN NUMBER,

  p_prescribed_date IN DATE,

  p_dosage IN VARCHAR2,

  p_frequency IN VARCHAR2

) IS

BEGIN

  INSERT INTO prescriptions (patient_id, medication_id, prescribed_date,
dosage, frequency)

  VALUES (p_patient_id, p_medication_id, p_prescribed_date, p_dosage,
p_frequency);

END;
```

Function: Get Medication Details

```
CREATE OR REPLACE FUNCTION get_medication_details(
  p_medication_id IN NUMBER
) RETURN VARCHAR2 IS
  v_medication_details VARCHAR2(4000);
BEGIN
  SELECT 'Medication: ' || medication_name || ', Dosage: ' || dosage || ',
Side Effects: ' || side_effects
  INTO v_medication_details
  FROM medications
  WHERE medication_id = p_medication_id;

  RETURN v_medication_details;
END;
```

Step 5: Example Usage

Adding a Patient

```
BEGIN
  patient_pkg.add_patient('John', 'Doe', TO_DATE('1980-01-01', 'YYYY-MM-DD'),
'Male', '123-456-7890');
END;
```

Updating Patient Contact

```
BEGIN
  patient_pkg.update_patient_contact(1, '987-654-3210');
END;
```

Getting Patient Details

```
DECLARE
  v_details VARCHAR2(4000);
BEGIN
  v_details := patient_pkg.get_patient_details(1);
  DBMS_OUTPUT.PUT_LINE(v_details);
```

```
END;
```

Adding a Prescription

```
BEGIN
  add_prescription(1, 1, SYSDATE, '500mg', 'Twice a day');
END;
```

Getting Medication Details

```
DECLARE
  v_medication_details VARCHAR2(4000);
BEGIN
  v_medication_details := get_medication_details(1);
  DBMS_OUTPUT.PUT_LINE(v_medication_details);
END;
```

Create an Audit Table

Table: Audit_Log

```
CREATE TABLE audit_log (
  audit_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  table_name VARCHAR2(50),
  operation VARCHAR2(10),
  record_id NUMBER,
  old_values CLOB,
  new_values CLOB,
  change_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Create Triggers for Logging Changes

Trigger for Patients Table

```
CREATE OR REPLACE TRIGGER trg_audit_patients
AFTER INSERT OR UPDATE OR DELETE ON patients
FOR EACH ROW
```

```
DECLARE

  v_old_values CLOB;

  v_new_values CLOB;

BEGIN

  IF INSERTING THEN

    v_new_values := 'first_name=' || :NEW.first_name || ', last_name='
|| :NEW.last_name || ', date_of_birth=' || TO_CHAR(:NEW.date_of_birth, 'YYYY-
MM-DD') || ', gender=' || :NEW.gender || ', contact_number='
|| :NEW.contact_number;

    INSERT INTO audit_log (table_name, operation, record_id, new_values)

    VALUES ('patients', 'INSERT', :NEW.patient_id, v_new_values);

  ELSIF UPDATING THEN

    v_old_values := 'first_name=' || :OLD.first_name || ', last_name='
|| :OLD.last_name || ', date_of_birth=' || TO_CHAR(:OLD.date_of_birth, 'YYYY-
MM-DD') || ', gender=' || :OLD.gender || ', contact_number='
|| :OLD.contact_number;

    v_new_values := 'first_name=' || :NEW.first_name || ', last_name='
|| :NEW.last_name || ', date_of_birth=' || TO_CHAR(:NEW.date_of_birth, 'YYYY-
MM-DD') || ', gender=' || :NEW.gender || ', contact_number='
|| :NEW.contact_number;

    INSERT INTO audit_log (table_name, operation, record_id, old_values,
new_values)

    VALUES ('patients', 'UPDATE', :NEW.patient_id, v_old_values,
v_new_values);

  ELSIF DELETING THEN

    v_old_values := 'first_name=' || :OLD.first_name || ', last_name='
|| :OLD.last_name || ', date_of_birth=' || TO_CHAR(:OLD.date_of_birth, 'YYYY-
MM-DD') || ', gender=' || :OLD.gender || ', contact_number='
|| :OLD.contact_number;

    INSERT INTO audit_log (table_name, operation, record_id, old_values)

    VALUES ('patients', 'DELETE', :OLD.patient_id, v_old_values);

  END IF;

END;
```

Trigger for Medications Table

```
CREATE OR REPLACE TRIGGER trg_audit_medications
```

```
AFTER INSERT OR UPDATE OR DELETE ON medications

FOR EACH ROW

DECLARE

  v_old_values CLOB;

  v_new_values CLOB;

BEGIN

  IF INSERTING THEN

    v_new_values := 'medication_name=' || :NEW.medication_name || ', dosage='
|| :NEW.dosage || ', side_effects=' || :NEW.side_effects;

    INSERT INTO audit_log (table_name, operation, record_id, new_values)

    VALUES ('medications', 'INSERT', :NEW.medication_id, v_new_values);

  ELSIF UPDATING THEN

    v_old_values := 'medication_name=' || :OLD.medication_name || ', dosage='
|| :OLD.dosage || ', side_effects=' || :OLD.side_effects;

    v_new_values := 'medication_name=' || :NEW.medication_name || ', dosage='
|| :NEW.dosage || ', side_effects=' || :NEW.side_effects;

    INSERT INTO audit_log (table_name, operation, record_id, old_values,
new_values)

    VALUES ('medications', 'UPDATE', :NEW.medication_id, v_old_values,
v_new_values);

  ELSIF DELETING THEN

    v_old_values := 'medication_name=' || :OLD.medication_name || ', dosage='
|| :OLD.dosage || ', side_effects=' || :OLD.side_effects;

    INSERT INTO audit_log (table_name, operation, record_id, old_values)

    VALUES ('medications', 'DELETE', :OLD.medication_id, v_old_values);

  END IF;

END;
```

Trigger for Prescriptions Table

```
CREATE OR REPLACE TRIGGER trg_audit_prescriptions

AFTER INSERT OR UPDATE OR DELETE ON prescriptions

FOR EACH ROW

DECLARE
```

```
  v_old_values CLOB;

  v_new_values CLOB;

BEGIN

  IF INSERTING THEN

    v_new_values := 'patient_id=' || :NEW.patient_id || ', medication_id='
|| :NEW.medication_id || ', prescribed_date=' ||
TO_CHAR(:NEW.prescribed_date, 'YYYY-MM-DD') || ', dosage=' || :NEW.dosage ||
', frequency=' || :NEW.frequency;

    INSERT INTO audit_log (table_name, operation, record_id, new_values)

    VALUES ('prescriptions', 'INSERT', :NEW.prescription_id, v_new_values);

  ELSIF UPDATING THEN

    v_old_values := 'patient_id=' || :OLD.patient_id || ', medication_id='
|| :OLD.medication_id || ', prescribed_date=' ||
TO_CHAR(:OLD.prescribed_date, 'YYYY-MM-DD') || ', dosage=' || :OLD.dosage ||
', frequency=' || :OLD.frequency;

    v_new_values := 'patient_id=' || :NEW.patient_id || ', medication_id='
|| :NEW.medication_id || ', prescribed_date=' ||
TO_CHAR(:NEW.prescribed_date, 'YYYY-MM-DD') || ', dosage=' || :NEW.dosage ||
', frequency=' || :NEW.frequency;

    INSERT INTO audit_log (table_name, operation, record_id, old_values,
new_values)

    VALUES ('prescriptions', 'UPDATE', :NEW.prescription_id, v_old_values,
v_new_values);

  ELSIF DELETING THEN

    v_old_values := 'patient_id=' || :OLD.patient_id || ', medication_id='
|| :OLD.medication_id || ', prescribed_date=' ||
TO_CHAR(:OLD.prescribed_date, 'YYYY-MM-DD') || ', dosage=' || :OLD.dosage ||
', frequency=' || :OLD.frequency;

    INSERT INTO audit_log (table_name, operation, record_id, old_values)

    VALUES ('prescriptions', 'DELETE', :OLD.prescription_id, v_old_values);

  END IF;

END;
```

Scheduling the Reports to run vi CRON.

To schedule the PL/SQL procedures to run one after the other using cron, and to ensure that if one job fails, the subsequent jobs do not run, you can follow these steps:

1. **Create Shell Scripts**: Create shell scripts to call the PL/SQL procedures.
2. **Create a Master Shell Script**: Create a master shell script to call each job sequentially and check for errors.
3. **Schedule the Master Shell Script Using Cron**: Schedule the master shell script using cron.

Step 1: Create Shell Scripts for Each Job

Shell Script for Generating Patient Report (generate_patient_report.sh**)**

```
#!/bin/bash

sqlplus -s username/password@database <<EOF
WHENEVER SQLERROR EXIT SQL.SQLCODE
BEGIN
  generate_patient_report;
END;
/
EXIT;
EOF

if [ $? -ne 0 ]; then
  echo "Error: generate_patient_report failed."
  exit 1
fi
```

Shell Script for Generating Prescription Report (generate_prescription_report.sh**)**

```
#!/bin/bash

sqlplus -s username/password@database <<EOF
WHENEVER SQLERROR EXIT SQL.SQLCODE
BEGIN
  generate_prescription_report;
END;
/
```

```
EXIT;
EOF

if [ $? -ne 0 ]; then
  echo "Error: generate_prescription_report failed."
  exit 1
fi
```

Step 2: Create a Master Shell Script

Master Shell Script (run_reports.sh**)**

```
#!/bin/bash

# Run generate_patient_report
./generate_patient_report.sh
if [ $? -ne 0 ]; then
  echo "Error: generate_patient_report failed. Stopping execution."
  exit 1
fi

# Run generate_prescription_report
./generate_prescription_report.sh
if [ $? -ne 0 ]; then
  echo "Error: generate_prescription_report failed. Stopping execution."
  exit 1
fi
echo "All reports generated successfully."
```

Step 3: Schedule the Master Shell Script Using Cron

Open the crontab file for editing:

crontab -e

Add a cron job to schedule the master shell script. For example, to run the script every day at 2 AM:

```
0 2 * * * /path/to/run_reports.sh >> /path/to/logfile.log 2>&1
```

Explanation:

Shell Scripts: Each shell script calls a PL/SQL procedure using sqlplus and checks for errors. If an error occurs, the script exits with a non-zero status.

Master Shell Script: The master shell script calls each job sequentially. If any job fails, the script stops execution and exits with a non-zero status.

Cron Job: The cron job schedules the master shell script to run at a specified time. The output is redirected to a log file for monitoring.

By following these steps, you can ensure that the PL/SQL procedures are executed sequentially, and if any job fails, the subsequent jobs do not run.

Summary

This simple Oracle PL/SQL application includes tables with relationships, triggers, packages, functions, and procedures for a hospital patient and medication tracking system. The application allows you to add and update patient information, manage prescriptions, and retrieve detailed information about patients and medications.

4. Advanced Error Handling

- **Logging and Auditing Errors**

Advanced Error Handling in Oracle PL/SQL

Advanced error handling in Oracle PL/SQL involves using various techniques to manage and respond to exceptions (errors) that occur during the execution of PL/SQL code. Proper error handling ensures that your code can gracefully handle unexpected situations and provide meaningful feedback to users or calling programs. This includes logging errors, providing detailed error messages, and using the DBMS_UTILITY package to show the traceback.

Key Techniques for Advanced Error Handling:

1. **Exception Handling**: Using EXCEPTION blocks to catch and handle exceptions.
2. **User-Defined Exceptions**: Creating custom exceptions for specific error conditions.
3. **RAISE_APPLICATION_ERROR**: Raising custom error messages with specific error codes.
4. **Logging Errors**: Recording error details in a log table for later analysis.
5. **Traceback**: Using the DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function to show the traceback to the calling code.
6. **Error Handling within Error Handling**: Handling errors that occur during the error handling process.

Step 1: Create an Error Log Table

Table: Error_Log

```
CREATE TABLE error_log (
  error_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  error_code NUMBER,
  error_message VARCHAR2(4000),
  error_stack VARCHAR2(4000),
  error_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create a Procedure to Log Errors

Procedure: Log_Error

```
CREATE OR REPLACE PROCEDURE log_error (
  p_error_code IN NUMBER,
  p_error_message IN VARCHAR2,
  p_error_stack IN VARCHAR2
) IS
BEGIN
  INSERT INTO error_log (error_code, error_message, error_stack)
  VALUES (p_error_code, p_error_message, p_error_stack);
EXCEPTION
  WHEN OTHERS THEN
    -- Handle errors that occur during error logging
    INSERT INTO error_log (error_code, error_message, error_stack)
    VALUES (-20001, 'Error occurred while logging error: ' || SQLERRM,
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END;
```

Step 3: Example of Advanced Error Handling in a Procedure

Procedure: Update_Salary

```
CREATE OR REPLACE PROCEDURE update_salary (
  p_employee_id IN NUMBER,
```

```
  p_new_salary IN NUMBER
) IS
  e_invalid_salary EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_invalid_salary, -20002);
BEGIN
  -- Validate the new salary
  IF p_new_salary < 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Invalid salary: ' || p_new_salary);
  END IF;

  -- Update the salary
  UPDATE employees
  SET salary = p_new_salary
  WHERE employee_id = p_employee_id;

  DBMS_OUTPUT.PUT_LINE('Salary updated for employee ID: ' || p_employee_id);
EXCEPTION
  WHEN e_invalid_salary THEN
    -- Handle user-defined exception
    log_error(SQLCODE, SQLERRM, DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
  WHEN OTHERS THEN
    -- Handle all other exceptions
    log_error(SQLCODE, SQLERRM, DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
    DBMS_OUTPUT.PUT_LINE('Unhandled error: ' || SQLERRM);
END;
```

Explanation:

1. **Error Log Table**: The error_log table is created to store error details, including the error code, error message, error stack, and timestamp.

2. **Log_Error Procedure**: The log_error procedure inserts error details into the error_log table. It also handles errors that occur during the error logging process.
3. **Update_Salary Procedure**: The update_salary procedure demonstrates advanced error handling. It includes:
    - **User-Defined Exception**: The e_invalid_salary exception is defined and initialized with a specific error code.
    - **RAISE_APPLICATION_ERROR**: The RAISE_APPLICATION_ERROR procedure raises a custom error if the new salary is invalid.
    - **Exception Handling**: The EXCEPTION block catches the e_invalid_salary exception and logs the error details using the log_error procedure. It also catches all other exceptions and logs their details.
    - **Traceback**: The DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function is used to show the traceback to the calling code.

Key Points to Remember:

- **Exception Handling**: Use EXCEPTION blocks to catch and handle specific exceptions.
- **User-Defined Exceptions**: Create custom exceptions for specific error conditions.
- **RAISE_APPLICATION_ERROR**: Use this procedure to raise custom error messages with specific error codes.
- **Logging Errors**: Record error details in a log table for later analysis.
- **Traceback**: Use the DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function to show the traceback to the calling code.

Error Handling in Autonomous Transactions

Error handling in autonomous transactions in Oracle PL/SQL is an important topic. Autonomous transactions allow you to perform independent transactions within a PL/SQL block, procedure, function, package, or trigger. These transactions are separate from the main transaction and can commit or roll back independently.

Key Points about Autonomous Transactions:

1. **Isolation**: Autonomous transactions are isolated from the main transaction. This means that changes made by the autonomous transaction are committed or rolled back independently of the main transaction.
2. **PRAGMA AUTONOMOUS_TRANSACTION**: This directive is used to declare a PL/SQL block as an autonomous transaction.
3. **Use Cases**: Common use cases include logging errors, auditing, and performing operations that should not be rolled back even if the main transaction fails.

Example of Error Handling in Autonomous Transactions:

Let's consider an example where we log errors using an autonomous transaction.

Step 1: Create an Error Log Table

```
SQL
CREATE TABLE error_log (
    id NUMBER(10) NOT NULL,
    log_timestamp TIMESTAMP NOT NULL,
    error_message VARCHAR2(4000)
);
```

Step 2: Create a Sequence for the Error Log Table

```
SQL
CREATE SEQUENCE error_log_seq
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;
```

Step 3: Create a Procedure to Log Errors

```
SQL
CREATE OR REPLACE PROCEDURE log_error (error_msg VARCHAR2) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO error_log (id, log_timestamp, error_message)
    VALUES (error_log_seq.NEXTVAL, SYSTIMESTAMP, error_msg);
    COMMIT; -- Commit the autonomous transaction
EXCEPTION
    WHEN OTHERS THEN
        -- Handle any unexpected errors
        ROLLBACK;
        RAISE;
END log_error;
/
```

Explanation:

- **PRAGMA AUTONOMOUS_TRANSACTION**: This directive makes the log_error procedure an autonomous transaction.
- **INSERT INTO error_log**: This statement inserts the error message into the error_log table.
- **COMMIT**: This commits the autonomous transaction, ensuring that the error log entry is saved even if the main transaction fails.
- **EXCEPTION**: The exception block handles any unexpected errors that occur within the autonomous transaction. If an error occurs, the transaction is rolled back, and the error is re-raised.

Important Considerations:

1. **Deadlocks**: Be cautious of potential deadlocks. If an autonomous transaction tries to access a resource held by the main transaction, a deadlock can occur[1].
2. **Error Handling**: Ensure that errors within the autonomous transaction are properly handled to avoid unexpected behavior[2].
3. **Commit or Rollback**: It is mandatory to either commit or roll back the autonomous transaction before it completes. If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception[3].

Additional Example: Nested Autonomous Transactions

Let's consider an example where we use nested autonomous transactions to update employee salaries and log the changes.

Step 1: Create an Employee Table

```
CREATE TABLE emp (
    emp_no NUMBER(10) NOT NULL,
    salary NUMBER(10, 2)
);
```

Step 2: Insert Sample Data

```
INSERT INTO emp (emp_no, salary) VALUES (1001, 50000);
INSERT INTO emp (emp_no, salary) VALUES (1002, 60000);
COMMIT;
```

Step 3: Create a Procedure to Update Salary and Log Changes

```
CREATE OR REPLACE PROCEDURE update_salary (emp_id NUMBER, increment NUMBER)
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
```

```
        UPDATE emp SET salary = salary + increment WHERE emp_no = emp_id;
        COMMIT; -- Commit the autonomous transaction

        -- Log the change
        log_error('Salary updated for employee ' || emp_id);
    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
            RAISE;
    END update_salary;
    /
```

Explanation:

- **PRAGMA AUTONOMOUS_TRANSACTION**: This directive makes the update_salary procedure an autonomous transaction.
- **UPDATE emp SET salary = salary + increment**: This statement updates the salary of the specified employee.
- **COMMIT**: This commits the autonomous transaction, ensuring that the salary update is saved.
- **log_error**: This calls the log_error procedure to log the change.

Conclusion

Autonomous transactions are powerful tools for error handling and logging in Oracle PL/SQL. They provide the flexibility to perform independent operations without affecting the main transaction. By using autonomous transactions, you can ensure that critical operations, such as error logging, are preserved even if the main transaction fails.

5. Advanced Cursors

Cursors in PL/SQL are used to handle query results. There are two main types of cursors:

1. **Implicit Cursors**: Automatically created by Oracle for single-row queries.
2. **Explicit Cursors**: Defined by the programmer for multi-row queries.

REF CURSORs

REF CURSORs, also known as cursor variables, provide a dynamic and flexible means to handle query results. Unlike explicit cursors, REF CURSORs are not tied to a specific query at compilation time. This allows you to open, fetch, and close them dynamically at runtime.

Types of REF CURSORs

1. **Strongly Typed REF CURSOR**: Associated with a specific structure or type.
2. **Weakly Typed REF CURSOR**: Not associated with any specific structure.

Example of REF CURSOR with Dynamic SQL

Let's consider an example where we use a REF CURSOR to dynamically fetch and display data from the employees table.

Step 1: Create the Employees Table

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50)
);

INSERT INTO employees VALUES (1, 'John Doe');
INSERT INTO employees VALUES (2, 'Jane Smith');
COMMIT;
```

Step 2: Declare and Use a REF CURSOR

```
DECLARE
    TYPE ref_cursor_type IS REF CURSOR;
    cursor_variable ref_cursor_type;
    emp_id employees.employee_id%TYPE;
    emp_name employees.employee_name%TYPE;
BEGIN
    -- Dynamic Query using REF CURSOR
    OPEN cursor_variable FOR 'SELECT employee_id, employee_name FROM
employees';

    -- Fetch and Display Data
```

```
    LOOP
        FETCH cursor_variable INTO emp_id, emp_name;
        EXIT WHEN cursor_variable%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id || ', Name: ' ||
emp_name);
    END LOOP;

    -- Close the Cursor
    CLOSE cursor_variable;
END;
/
```

Explanation:

- **TYPE ref_cursor_type IS REF CURSOR**: Declares a REF CURSOR type.
- **cursor_variable ref_cursor_type**: Declares a cursor variable of the REF CURSOR type.
- **OPEN cursor_variable FOR 'SELECT employee_id, employee_name FROM employees'**: Opens the cursor for a dynamic SQL query.
- **FETCH cursor_variable INTO emp_id, emp_name**: Fetches data from the cursor into variables.
- **DBMS_OUTPUT.PUT_LINE**: Displays the fetched data.
- **CLOSE cursor_variable**: Closes the cursor.

Using REF CURSORs in Procedures

You can also use REF CURSORs in procedures to pass query results between different program units.

Example: Procedure with REF CURSOR

```
CREATE OR REPLACE PROCEDURE get_employees (emp_cursor OUT SYS_REFCURSOR) IS
BEGIN
    OPEN emp_cursor FOR 'SELECT employee_id, employee_name FROM employees';
END;
/
```

Calling the Procedure

```
DECLARE
    emp_cursor SYS_REFCURSOR;
    emp_id employees.employee_id%TYPE;
    emp_name employees.employee_name%TYPE;
BEGIN
    -- Call the Procedure
    get_employees(emp_cursor);

    -- Fetch and Display Data
    LOOP
        FETCH emp_cursor INTO emp_id, emp_name;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id || ', Name: ' ||
emp_name);
    END LOOP;

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

Explanation:

- **SYS_REFCURSOR**: A predefined weakly typed REF CURSOR.
- **get_employees**: A procedure that opens a REF CURSOR for a dynamic SQL query.
- **emp_cursor OUT SYS_REFCURSOR**: An OUT parameter to pass the REF CURSOR.
- **FETCH emp_cursor INTO emp_id, emp_name**: Fetches data from the cursor into variables.
- **DBMS_OUTPUT.PUT_LINE**: Displays the fetched data.
- **CLOSE emp_cursor**: Closes the cursor.

Conclusion

REF CURSORs provide a powerful way to handle dynamic SQL queries and pass query results between different PL/SQL program units. They offer flexibility and reusability, making them an essential tool for advanced PL/SQL programming.

Advanced Cursor Attributes

Let's explore advanced cursor attributes in Oracle PL/SQL, including %ROWTYPE and %TYPE with cursors.

Cursor Attributes

Cursor attributes provide information about the state of the cursor. The main cursor attributes are:

1. **%ISOPEN**: Returns TRUE if the cursor is open, FALSE otherwise.
2. **%FOUND**: Returns TRUE if the last fetch returned a row, FALSE otherwise.
3. **%NOTFOUND**: Returns TRUE if the last fetch did not return a row, FALSE otherwise.
4. **%ROWCOUNT**: Returns the number of rows fetched so far.

Using %ROWTYPE **and** %TYPE **with Cursors**

- **%ROWTYPE**: This attribute is used to declare a record that can hold an entire row of data from a table or cursor.
- **%TYPE**: This attribute is used to declare a variable with the same data type as a column in a table or a field in a record.

Example: Using Cursor Attributes with %ROWTYPE **and** %TYPE

Let's consider an example where we use these attributes with a cursor to fetch and display employee data.

Step 1: Create the Employees Table

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    salary NUMBER(10, 2)
);


INSERT INTO employees VALUES (1, 'John Doe', 50000);
INSERT INTO employees VALUES (2, 'Jane Smith', 60000);
COMMIT;
```

Step 2: Declare and Use a Cursor with %ROWTYPE **and** %TYPE

```
DECLARE
```

```
    -- Declare a cursor
    CURSOR emp_cursor IS
        SELECT employee_id, employee_name, salary FROM employees;

    -- Declare a record using %ROWTYPE
    emp_record employees%ROWTYPE;

    -- Declare variables using %TYPE
    emp_id employees.employee_id%TYPE;
    emp_name employees.employee_name%TYPE;
    emp_salary employees.salary%TYPE;
BEGIN
    -- Open the cursor
    OPEN emp_cursor;

    -- Fetch data into the record
    LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;

        -- Display data using the record
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.employee_id ||
                             ', Name: ' || emp_record.employee_name ||
                             ', Salary: ' || emp_record.salary);
    END LOOP;

    -- Close the cursor
    CLOSE emp_cursor;
```

```
    -- Open the cursor again
    OPEN emp_cursor;

    -- Fetch data into individual variables
    LOOP
        FETCH emp_cursor INTO emp_id, emp_name, emp_salary;
        EXIT WHEN emp_cursor%NOTFOUND;

        -- Display data using individual variables
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id ||
                             ', Name: ' || emp_name ||
                             ', Salary: ' || emp_salary);
    END LOOP;

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

Explanation:

- **CURSOR emp_cursor IS SELECT …**: Declares a cursor to fetch employee data.
- **emp_record employees%ROWTYPE**: Declares a record that can hold an entire row of data from the employees table.
- **emp_id employees.employee_id%TYPE**: Declares a variable with the same data type as the employee_id column.
- **OPEN emp_cursor**: Opens the cursor.
- **FETCH emp_cursor INTO emp_record**: Fetches data into the record.
- **DBMS_OUTPUT.PUT_LINE**: Displays the fetched data.
- **CLOSE emp_cursor**: Closes the cursor.
- **FETCH emp_cursor INTO emp_id, emp_name, emp_salary**: Fetches data into individual variables.

Conclusion

Using cursor attributes along with %ROWTYPE and %TYPE provides a flexible and efficient way to handle query results in Oracle PL/SQL. These attributes help you manage and manipulate data with ease, ensuring that your PL/SQL programs are robust and maintainable.

5. Advanced Functions and Procedures.

Pipelining in PL/SQL

Pipelined table functions allow you to return rows to the calling query as they are produced, rather than waiting for the entire function to complete. This can improve performance, especially for large datasets.

Example: Pipelined Table Function

Let's create a pipelined table function that generates a series of numbers.

Step 1: Create the Types

```
CREATE TYPE number_row AS OBJECT (
    num NUMBER
);
CREATE TYPE number_table AS TABLE OF number_row;
```

Step 2: Create the Pipelined Table Function

```
CREATE OR REPLACE FUNCTION generate_numbers (p_limit NUMBER)
RETURN number_table PIPELINED IS
BEGIN
    FOR i IN 1 .. p_limit LOOP
        PIPE ROW (number_row(i));
    END LOOP;
    RETURN;
END generate_numbers;
/
```

Step 3: Query the Pipelined Table Function

```
SELECT * FROM TABLE(generate_numbers(10));
```

Explanation:

- **CREATE TYPE**: Defines the object and table types.
- **PIPELINED**: Indicates that the function will return rows as they are produced.
- **PIPE ROW**: Sends a row to the calling query.

- **RETURN**: Ends the function.

Advanced Overloading in PL/SQL

Overloading allows you to define multiple procedures or functions with the same name but different parameter lists. This can make your code more flexible and easier to use.

Example: Advanced Overloading

Let's create a package with overloaded procedures.

Step 1: Create the Package Specification

```
CREATE OR REPLACE PACKAGE employee_pkg IS

    PROCEDURE update_employee (p_emp_id NUMBER, p_salary NUMBER);

    PROCEDURE update_employee (p_emp_id NUMBER, p_job_id VARCHAR2);

    PROCEDURE update_employee (p_emp_id NUMBER, p_salary NUMBER, p_job_id
VARCHAR2);

END employee_pkg;

/
```

Step 2: Create the Package Body

```
CREATE OR REPLACE PACKAGE BODY employee_pkg IS

    PROCEDURE update_employee (p_emp_id NUMBER, p_salary NUMBER) IS

    BEGIN

        UPDATE employees SET salary = p_salary WHERE employee_id = p_emp_id;

    END update_employee;


    PROCEDURE update_employee (p_emp_id NUMBER, p_job_id VARCHAR2) IS

    BEGIN

        UPDATE employees SET job_id = p_job_id WHERE employee_id = p_emp_id;

    END update_employee;


    PROCEDURE update_employee (p_emp_id NUMBER, p_salary NUMBER, p_job_id
VARCHAR2) IS

    BEGIN

        UPDATE employees SET salary = p_salary, job_id = p_job_id WHERE
employee_id = p_emp_id;
```

```
    END update_employee;

END employee_pkg;

/
```

Explanation:

- **Package Specification**: Declares the overloaded procedures.
- **Package Body**: Implements the overloaded procedures.
- **update_employee**: Overloaded procedures to update employee details based on different parameters.

Conclusion

Pipelining and advanced overloading are powerful techniques in Oracle PL/SQL that can enhance the performance and flexibility of your procedures and functions. Pipelined table functions allow for efficient data processing, while overloading enables you to create versatile and reusable code.

6. **Advanced Packages**
   Here is an example of calling the above packages demonstrating the global variable keeping state between session calls.

   Calling the emp_pkg **Package**

   Example 1: Basic Package with Global Variables and Initialization Section

   1. **Increment Employee Count**:

      ```
      BEGIN

          emp_pkg.increment_emp_count;

          emp_pkg.increment_emp_count;

          emp_pkg.increment_emp_count;

      END;

      /
      ```

   2. **Get Employee Count**:

      ```
      DECLARE

          emp_count NUMBER;

      BEGIN

          emp_count := emp_pkg.get_emp_count;

          DBMS_OUTPUT.PUT_LINE('Employee Count: ' || emp_count);

      END;

      /
      ```

Explanation:

- **emp_pkg.increment_emp_count**: Calls the increment_emp_count procedure to increment the global employee count.
- **emp_pkg.get_emp_count**: Calls the get_emp_count function to retrieve the current employee count.

Example 2: Advanced Implementation with Audit Trail

1. **Increment Employee Count and Log Action**:

```
BEGIN
    emp_pkg.increment_emp_count;
END;
/
```

2. **Audit Action**:

```
BEGIN
    emp_pkg.audit_action(1, 'Test Action');
END;
/
```

3. **Get Employee Count and Display Audit Log**:

```
DECLARE
    emp_count NUMBER;
BEGIN
    emp_count := emp_pkg.get_emp_count;
    DBMS_OUTPUT.PUT_LINE('Employee Count: ' || emp_count);

    FOR rec IN (SELECT * FROM emp_audit ORDER BY action_timestamp) LOOP
        DBMS_OUTPUT.PUT_LINE('Audit ID: ' || rec.audit_id ||
                             ', Employee ID: ' || rec.emp_id ||
                             ', Action: ' || rec.action ||
                             ', Timestamp: ' || rec.action_timestamp);
    END LOOP;
END;
```

/

Explanation:

- **emp_pkg.increment_emp_count**: Calls the increment_emp_count procedure to increment the global employee count and log the action.
- **emp_pkg.audit_action**: Calls the audit_action procedure to log a test action.
- **emp_pkg.get_emp_count**: Calls the get_emp_count function to retrieve the current employee count.
- **SELECT * FROM emp_audit**: Retrieves and displays the audit log.

These examples demonstrate how to call the procedures and functions within the emp_pkg package. You can use these calls to manage employee data and maintain an audit trail.

8. Advanced Triggers in Oracle PL/SQL

Compound Triggers

Compound triggers were introduced in Oracle 11g to address issues like mutating tables and to simplify the management of multiple related triggers. A compound trigger allows you to define actions for different timing points within a single trigger.

A compound trigger can have the following sections:

1. **Before Statement**: Executes before the triggering statement.
2. **Before Each Row**: Executes before each row affected by the triggering statement.
3. **After Each Row**: Executes after each row affected by the triggering statement.
4. **After Statement**: Executes after the triggering statement.

Example: Compound Trigger

Let's create a compound trigger to log changes to an employees table.

```
CREATE TABLE emp_log (
    log_timestamp TIMESTAMP NOT NULL,
    change_type VARCHAR2(1) NOT NULL,
    employee_id INTEGER NOT NULL
);


CREATE OR REPLACE TRIGGER emp_ct
FOR INSERT OR UPDATE OR DELETE ON employees
COMPOUND TRIGGER
```

```
    TYPE emp_t IS TABLE OF employees.employee_id%TYPE INDEX BY
BINARY_INTEGER;
    emps emp_t;

    BEFORE STATEMENT IS
    BEGIN
        NULL; -- Placeholder for any initialization code
    END BEFORE STATEMENT;

    BEFORE EACH ROW IS
    BEGIN
        NULL; -- Placeholder for any row-level code before the change
    END BEFORE EACH ROW;

    AFTER EACH ROW IS
    BEGIN
        emps(emps.COUNT + 1) := :NEW.employee_id;
    END AFTER EACH ROW;

    AFTER STATEMENT IS
    BEGIN
        FORALL i IN 1 .. emps.COUNT
            INSERT INTO emp_log (log_timestamp, change_type, employee_id)
            VALUES (SYSTIMESTAMP, 'I', emps(i));
    END AFTER STATEMENT;
END emp_ct;
/
```

Explanation:

- **BEFORE STATEMENT**: Executes before the triggering statement.
- **BEFORE EACH ROW**: Executes before each row affected by the triggering statement.

- **AFTER EACH ROW**: Collects the affected employee IDs.
- **AFTER STATEMENT**: Logs the changes to the emp_log table.

Trigger Performance Tuning

Performance tuning for triggers involves optimizing the trigger code to minimize overhead and improve efficiency. Here are some tips for tuning triggers:

1. **Minimize SQL and PL/SQL Context Switching**: Reduce the number of context switches between SQL and PL/SQL by using bulk operations like FORALL and BULK COLLECT.
2. **Use Conditional Predicates**: Use the WHEN clause to limit the execution of the trigger to specific conditions.
3. **Avoid Complex Logic**: Keep the trigger logic simple and avoid complex computations within the trigger.
4. **Use Compound Triggers**: Combine multiple related triggers into a single compound trigger to reduce overhead and avoid mutating table errors.

Example: Performance Tuning with Conditional Predicates

Let's modify the previous example to include a conditional predicate.

```
CREATE OR REPLACE TRIGGER emp_ct

FOR INSERT OR UPDATE OR DELETE ON employees

COMPOUND TRIGGER

    TYPE emp_t IS TABLE OF employees.employee_id%TYPE INDEX BY
BINARY_INTEGER;

    emps emp_t;


    BEFORE STATEMENT IS

    BEGIN

        NULL; -- Placeholder for any initialization code

    END BEFORE STATEMENT;


    BEFORE EACH ROW IS

    BEGIN

        NULL; -- Placeholder for any row-level code before the change

    END BEFORE EACH ROW;
```

```
    AFTER EACH ROW IS
    BEGIN
        IF INSERTING THEN
            emps(emps.COUNT + 1) := :NEW.employee_id;
        END IF;
    END AFTER EACH ROW;

    AFTER STATEMENT IS
    BEGIN
        FORALL i IN 1 .. emps.COUNT
            INSERT INTO emp_log (log_timestamp, change_type, employee_id)
            VALUES (SYSTIMESTAMP, 'I', emps(i));
    END AFTER STATEMENT;
END emp_ct;
/
```

Explanation:

- **IF INSERTING**: The conditional predicate ensures that the trigger only logs insert operations, reducing unnecessary processing.

Conclusion

Advanced triggers, including compound triggers, provide powerful tools for managing complex business logic in Oracle PL/SQL. By following performance tuning best practices, you can ensure that your triggers run efficiently and effectively.

9. Advanced Performance Tuning in Oracle PL/SQL and SQL

In-Depth Tips for Optimizing PL/SQL Code

1. **Use Bulk Operations**:
    - **FORALL**: Use the FORALL statement to perform bulk DML operations, which reduces context switching between PL/SQL and SQL engines.
    - **BULK COLLECT**: Use BULK COLLECT to fetch multiple rows at once into PL/SQL collections, improving performance by reducing the number of context switches.

```
DECLARE
    TYPE emp_tab IS TABLE OF employees%ROWTYPE;
```

```
    l_emps emp_tab;
BEGIN
    SELECT * BULK COLLECT INTO l_emps FROM employees;
    FORALL i IN l_emps.FIRST .. l_emps.LAST
        INSERT INTO employees_backup VALUES l_emps(i);
END;
/
```

2. **Minimize Context Switching**:
    - Reduce the number of SQL statements executed within PL/SQL loops.
    - Use PL/SQL collections to process data in bulk.
3. **Optimize SQL Statements**:
    - Use appropriate indexes to speed up query execution.
    - Avoid unnecessary columns in SELECT statements.
    - Use bind variables to reduce parsing overhead.
4. **Use PL/SQL Compiler Optimizations**:
    - Set the PLSQL_OPTIMIZE_LEVEL parameter to control the level of optimization applied by the PL/SQL compiler.
    - Use the PRAGMA INLINE directive to inline frequently called subprograms.

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;
```

5. **Avoid Implicit Data Type Conversions**:
    - Ensure that data types match between PL/SQL variables and table columns to avoid implicit conversions.
6. **Use Native Compilation**:
    - Compile PL/SQL code to native machine code to improve execution speed.

```
ALTER SESSION SET plsql_code_type = NATIVE;
```

Native compilation in Oracle PL/SQL converts PL/SQL code into native machine code, which can significantly improve the performance of PL/SQL programs. This process involves translating PL/SQL code into C code, compiling it with a C compiler, and linking it into the Oracle process as shared libraries.

Steps for Native Compilation

Step 1: Set Initialization Parameters

Before you can use native compilation, you need to set the following initialization parameters:

```
   ALTER SYSTEM SET plsql_native_library_dir = '/path/to/native/library';
   ALTER SYSTEM SET plsql_native_make_utility = 'make';
   ALTER SYSTEM SET plsql_native_make_file_name = '/path/to/makefile.mk';
```

Step 2: Enable Native Compilation

You can enable native compilation at the session or system level:

```
   ALTER SESSION SET plsql_code_type = NATIVE;
-- or
   ALTER SYSTEM SET plsql_code_type = NATIVE;
```

Step 3: Compile PL/SQL Code

Compile your PL/SQL code to native:

```
ALTER PROCEDURE your_procedure COMPILE;
-- or
ALTER PACKAGE your_package COMPILE;
```

Example: Native Compilation of a PL/SQL Procedure

Step 1: Create a Sample Procedure

Let's create a simple PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE test_speed AS
    v_number NUMBER;
BEGIN
    FOR i IN 1 .. 1000000 LOOP
        v_number := i / 1000;
    END LOOP;
END;
/
```

Step 2: Compile the Procedure in Interpreted Mode

First, compile and execute the procedure in interpreted mode:

```
   ALTER SESSION SET plsql_code_type = INTERPRETED;
   ALTER PROCEDURE test_speed COMPILE;
```

```
SET TIMING ON;

EXEC test_speed;

SET TIMING OFF;
```

Step 3: Compile the Procedure in Native Mode

Next, compile and execute the procedure in native mode:

```
ALTER SESSION SET plsql_code_type = NATIVE;

ALTER PROCEDURE test_speed COMPILE;


SET TIMING ON;

EXEC test_speed;

SET TIMING OFF;
```

Performance Comparison

By comparing the execution times of the procedure in interpreted and native modes, you can observe the performance improvements. Typically, native compilation can significantly speed up procedural logic, such as loops and calculations, but may not have a substantial impact on database calls[1].

Additional Considerations

- Compatibility: Ensure that your environment is compatible with native compilation, including the availability of a C compiler.
- Maintenance: Native compilation can complicate maintenance and debugging, so use it judiciously.
- Mixed Mode: You can mix native and interpreted code within the same database, but be aware of potential performance implications.

Tools for Performance Analysis

In addition to native compilation, you can use tools like SQL Trace and TKPROF for performance analysis:

- **SQL Trace: Enables detailed tracing of SQL execution.**
- **TKPROF: Formats trace files into readable reports for performance analysis.**

```
-- Enable SQL Trace

ALTER SESSION SET sql_trace = TRUE;


-- Run your PL/SQL code
```

```
-- Disable SQL Trace

ALTER SESSION SET sql_trace = FALSE;

-- Use TKPROF to analyze the trace file

tkprof tracefile.trc outputfile.prf EXPLAIN=username/password
```

TKPROF is a complex tool used to analyze SQL statements and a complete explanation of the tool is beyond the scope of this booklet. But here are some recommended websites.

Here are a few good websites to learn how to use TKPROF:

1. **Expert Oracle Blog**:
    - This blog has a comprehensive series on performance tuning basics, including detailed posts on using SQL Trace and TKPROF. The series is divided into multiple parts, covering tracing methods, generating TKPROF files, and analyzing TKPROF files.
    - Part 1: Trace and the methods to generate trace
    - Part 2: Generating TKPROF from trace file
    - Part 3: Analyzing TKPROF files
2. **Oracle Documentation**:
    - The official Oracle documentation provides detailed information on using SQL Trace and TKPROF, including examples and best practices.
    - Oracle Database Performance Tuning Guide
3. **Oracle Base**:
    - Oracle Base offers tutorials and articles on various Oracle topics, including a detailed guide on using TKPROF for performance tuning.
    - Oracle Base: TKPROF

These resources should help you get a good understanding of how to use TKPROF for performance analysis and tuning in Oracle PL/SQL.

Recommended Tools

- **Oracle Enterprise Manager (OEM): Comprehensive monitoring and performance tuning.**
- **SQL Developer: Includes SQL Tuning Advisor and SQL Monitor.**
- **AWR (Automatic Workload Repository): Collects and maintains performance statistics.**
- **ASH (Active Session History): Real-time monitoring of active sessions.**
- **SolarWinds Database Performance Analyzer: Third-party tool for detailed performance analysis.**

Using SQL Trace and TKPROF – a brief overview.

SQL Trace

The SQL Trace facility provides detailed performance information about SQL statements executed by an application. It generates trace files containing statistics such as parse, execute, and fetch counts, CPU and elapsed times, and more.

1. **Enable SQL Trace**:

```
ALTER SESSION SET sql_trace = TRUE;
```

2. **Run the Application**:
    - Execute the SQL statements or PL/SQL blocks you want to trace.
3. **Disable SQL Trace**:

```
ALTER SESSION SET sql_trace = FALSE;
```

TKPROF

TKPROF is a utility that formats the trace file generated by SQL Trace into a readable report.

1. **Run TKPROF**:

```
tkprof tracefile.trc outputfile.prf EXPLAIN=username/password
```

2. **Analyze the Report**:
    - The TKPROF report provides detailed information about the execution of SQL statements, including execution plans, resource usage, and more.

Additional Tools for Performance Analysis

1. **Oracle Enterprise Manager (OEM)**:
    - Provides comprehensive monitoring and performance tuning capabilities for Oracle databases.
2. **SQL Developer**:
    - Includes features for tuning SQL queries, such as the SQL Tuning Advisor and SQL Monitor.
3. **AWR (Automatic Workload Repository)**:
    - Collects, processes, and maintains performance statistics for problem detection and tuning.
4. **ASH (Active Session History)**:
    - Provides real-time monitoring of active sessions and helps identify performance bottlenecks.
5. **SolarWinds Database Performance Analyzer**:

- A third-party tool that provides detailed performance analysis and tuning recommendations for Oracle databases.

6. **RedGate Deployment Suite for Oracle**:
    - Includes tools for schema comparison, synchronization, and performance monitoring.

By leveraging these advanced techniques and tools, you can significantly improve the performance of your Oracle PL/SQL and SQL code, ensuring efficient and optimized database operations.

10. Security

Fine-grained access control (FGAC) in Oracle Database, also known as Virtual Private Database (VPD), allows you to enforce security policies at the row and column level. This means you can control access to specific rows and columns of a table based on the user's identity or other criteria.

Key Concepts of Fine-Grained Access Control

1. **Policies**: Security policies are defined using PL/SQL functions that return a predicate (WHERE clause) to be applied to SQL statements.
2. **Application Context**: Application contexts store session-specific information that can be used in security policies.
3. **DBMS_RLS Package**: This package is used to manage FGAC policies.

Example: Implementing Fine-Grained Access Control

Let's create an example to demonstrate FGAC. We'll use an employees table and restrict access based on the department.

Step 1: Create the Employees Table

```
CREATE TABLE employees (
    employee_id NUMBER PRIMARY KEY,
    employee_name VARCHAR2(50),
    department_id NUMBER
);


INSERT INTO employees VALUES (1, 'John Doe', 10);
INSERT INTO employees VALUES (2, 'Jane Smith', 20);
INSERT INTO employees VALUES (3, 'Alice Johnson', 10);
INSERT INTO employees VALUES (4, 'Bob Brown', 30);
COMMIT;
```

Step 2: Create an Application Context

```
CREATE OR REPLACE CONTEXT emp_ctx USING emp_ctx_pkg;
```

Step 3: Create a PL/SQL Package to Set the Context

```
CREATE OR REPLACE PACKAGE emp_ctx_pkg IS

    PROCEDURE set_dept_context(p_dept_id NUMBER);

END emp_ctx_pkg;

/


CREATE OR REPLACE PACKAGE BODY emp_ctx_pkg IS

    PROCEDURE set_dept_context(p_dept_id NUMBER) IS

    BEGIN

        DBMS_SESSION.SET_CONTEXT('emp_ctx', 'dept_id', p_dept_id);

    END set_dept_context;

END emp_ctx_pkg;

/
```

Step 4: Create a Security Policy Function

SQL

```
CREATE OR REPLACE FUNCTION emp_policy_function (schema_name VARCHAR2,
table_name VARCHAR2)

RETURN VARCHAR2 IS

    l_predicate VARCHAR2(200);

BEGIN

    l_predicate := 'department_id = SYS_CONTEXT(''emp_ctx'', ''dept_id'')';

    RETURN l_predicate;

END emp_policy_function;

/
```

Step 5: Add the Security Policy to the Table

```
BEGIN

    DBMS_RLS.ADD_POLICY(

        object_schema   => 'HR',
```

```
        object_name     => 'employees',
        policy_name     => 'emp_policy',
        function_schema => 'HR',
        policy_function => 'emp_policy_function'
    );
END;
/
```

Explanation:

- **Application Context**: emp_ctx is created to store the department ID.
- **PL/SQL Package**: emp_ctx_pkg sets the department ID in the context.
- **Security Policy Function**: emp_policy_function returns a predicate to filter rows based on the department ID.
- **DBMS_RLS.ADD_POLICY**: Adds the security policy to the employees table.

Using the Fine-Grained Access Control

```
Step 1: Set the Context
BEGIN
    emp_ctx_pkg.set_dept_context(10);
END;
/
```

Step 2: Query the Employees Table

```
SELECT * FROM employees;
```

Expected Output:

```
EMPLOYEE_ID  EMPLOYEE_NAME  DEPARTMENT_ID
-----------  -------------  -------------
1            John Doe       10
3            Alice Johnson  10
```

Conclusion

Fine-grained access control in Oracle Database allows you to enforce security policies at a granular level. By using application contexts and security policy functions, you can dynamically control access to data based on user-specific criteria.

11. Best Practices

Advanced Code Review Techniques in Oracle SQL and PL/SQL with a Team

Code reviews are essential for maintaining code quality, ensuring best practices, and fostering knowledge sharing within a team. Here are some advanced techniques for conducting effective code reviews in Oracle SQL and PL/SQL:

1. Establish Clear Guidelines

- **Coding Standards**: Define and document coding standards for SQL and PL/SQL. This includes naming conventions, formatting, and best practices.
- **Review Checklist**: Create a checklist of common issues to look for during reviews, such as performance bottlenecks, security vulnerabilities, and adherence to coding standards.

2. Use Collaborative Tools

- **Version Control Systems**: Use version control systems like Git to manage code changes and facilitate collaborative reviews.
- **Code Review Tools**: Utilize tools like GitHub, GitLab, or Bitbucket for code reviews. These platforms provide features like inline comments, pull requests, and approval workflows.

3. Conduct Regular Reviews

- **Scheduled Reviews**: Schedule regular code review sessions to ensure timely feedback and continuous improvement.
- **Peer Reviews**: Encourage peer reviews where team members review each other's code. This promotes knowledge sharing and helps identify issues early.

4. Focus on Key Areas

- **Performance**: Check for efficient SQL queries, proper indexing, and use of bulk operations in PL/SQL.
- **Security**: Ensure that the code is free from SQL injection vulnerabilities and follows security best practices.
- **Maintainability**: Evaluate the code for readability, modularity, and adherence to design principles.

5. Provide Constructive Feedback

- **Positive Reinforcement**: Highlight good practices and well-written code.
- **Actionable Suggestions**: Provide clear and actionable suggestions for improvement.
- **Collaborative Discussion**: Encourage open discussions to understand the rationale behind code decisions and explore alternative solutions.

Refactoring and Improving Legacy PL/SQL Code

Refactoring legacy code involves improving the structure and readability of existing code without changing its functionality. Here are some steps and best practices for refactoring and improving legacy PL/SQL code:

1. Analyze and Understand the Code

- **Code Review**: Conduct a thorough review of the legacy code to identify areas for improvement.
- **Documentation**: Ensure that the code is well-documented, including comments and explanations of complex logic.

2. Identify Refactoring Opportunities

- **Redundant Code**: Remove redundant or duplicate code to simplify the codebase.
- **Modularization**: Break down large procedures and functions into smaller, reusable modules.
- **Naming Conventions**: Use meaningful and consistent naming conventions for variables, procedures, and functions.

3. Implement Refactoring Techniques

- **Extract Method**: Extract complex logic into separate procedures or functions to improve readability and maintainability.
- **Replace Loops with Bulk Operations**: Use bulk operations like FORALL and BULK COLLECT to improve performance.
- **Use Collections**: Replace individual variable declarations with collections to handle multiple values efficiently.

4. Test and Validate

- **Unit Testing**: Write unit tests to validate the functionality of the refactored code.
- **Regression Testing**: Perform regression testing to ensure that the refactoring does not introduce new issues.

5. Continuous Improvement

- **Code Reviews**: Regularly review the refactored code to identify further improvement opportunities.
- **Performance Monitoring**: Monitor the performance of the refactored code and make adjustments as needed.

Example: Refactoring a Legacy PL/SQL Procedure

Original Procedure

```
CREATE OR REPLACE PROCEDURE process_employees IS
    CURSOR emp_cur IS SELECT employee_id, salary FROM employees;
    v_emp_id employees.employee_id%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cur;
    LOOP
```

```
        FETCH emp_cur INTO v_emp_id, v_salary;

        EXIT WHEN emp_cur%NOTFOUND;

        -- Process each employee

        UPDATE employees SET salary = v_salary * 1.1 WHERE employee_id =
v_emp_id;

    END LOOP;

    CLOSE emp_cur;

END;

/
```

Refactored Procedure

```
CREATE OR REPLACE PROCEDURE process_employees IS

    TYPE emp_tab IS TABLE OF employees%ROWTYPE;

    l_emps emp_tab;

BEGIN

    SELECT * BULK COLLECT INTO l_emps FROM employees;

    FORALL i IN l_emps.FIRST .. l_emps.LAST

        UPDATE employees SET salary = l_emps(i).salary * 1.1 WHERE
employee_id = l_emps(i).employee_id;

END;

/
```

Explanation:

- **Bulk Collect**: The refactored procedure uses BULK COLLECT to fetch all employee records into a collection.
- **FORALL**: The FORALL statement is used to perform bulk updates, reducing context switching and improving performance.

Additional Resources

- **Oracle Magazine**: Refactoring for PL/SQL Developers
- **ModLogix**: Refactoring Legacy Code: Steps and Best Practices
- **dbt Developer Hub**: Refactoring Legacy SQL to dbt

By following these advanced code review techniques and refactoring practices, you can improve the quality, performance, and maintainability of your Oracle SQL and PL/SQL code.

12. Resources for Further Learning

Intermediate Level Books:

1. **"Oracle PL/SQL Programming" by Steven Feuerstein and Bill Pribyl**: This book is a comprehensive guide to PL/SQL programming and covers advanced topics in detail.
2. **"Oracle PL/SQL Best Practices" by Steven Feuerstein**: This book provides practical advice and best practices for writing efficient and maintainable PL/SQL code.
3. **"Oracle PL/SQL Programming: A Developer's Workbook" by Steven Feuerstein and Andrew Odewahn**: This workbook offers exercises and solutions to help you practice and master PL/SQL concepts.

Recommended Reading:

1. **Oracle Documentation**: The official Oracle documentation is an excellent resource for learning about PL/SQL features and best practices. You can find it on the Oracle website.
2. **"Mastering Oracle PL/SQL: Practical Solutions" by Christopher Beck and Lex deHaan**: This book provides practical solutions and advanced techniques for PL/SQL programming.

Online Courses:

1. **Udemy**: Courses like "The Complete PL/SQL Bootcamp: Beginner to Advanced PL/SQL" offer comprehensive training on PL/SQL.
2. **Coursera**: The "Oracle PL/SQL Fundamentals" course by Oracle provides a solid foundation and covers intermediate topics.
3. **Pluralsight**: Offers various courses on PL/SQL, including advanced topics and best practices.

Platforms for Intermediate Learning:

1. **Oracle Learning Library**: Offers tutorials, videos, and hands-on labs for learning PL/SQL.
2. **LinkedIn Learning**: Provides courses on PL/SQL programming and database management.
3. **edX**: Offers courses on Oracle PL/SQL and database management from top universities and institutions.

Communities:

1. **Oracle Community**: The official Oracle forums are a great place to ask questions, share knowledge, and connect with other Oracle professionals. Oracle Community
2. **Stack Overflow**: A popular platform for asking and answering technical questions, including PL/SQL-related queries.
3. **Reddit**: Subreddits like r/Oracle and r/SQL are good places to discuss PL/SQL topics and get help from the community.

Where to Get Help:

1. **Oracle Support**: If you have an Oracle support contract, you can access Oracle Support for technical assistance.
2. **Local User Groups**: Joining local Oracle user groups can provide opportunities for networking and learning from other professionals.
3. **Meetups and Conferences**: Attend Oracle-related meetups and conferences to learn from experts and stay updated on the latest trends and best practices.