Chapter 6 - Handling Different Objects in SQL & PL-SQL

PL/SQL (Procedural Language/Structured Query Language)

Introduction

PL/SQL **(Procedural Language/Structured Query Language)** is Oracle Corporation's proprietary procedural extension to SQL, the standard database query language. PL/SQL is one of the core technologies at Oracle and is essential to leveraging the full potential of Oracle Database.

PL/SQL combines the relational data access capabilities of the Structured Query Language with a flexible embedded procedural language, and it executes complex queries and programmatic logic run inside the database engine itself. This enhances the agility, efficiency, and performance of database-driven applications.

Basics of PL/SQL

1. PL/SQL Block Structure:

It introduces you to PL/SQL block structure and shows you how to develop the first running PL/SQL program. PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

```
[DECLARE]
        Declaration statements;
BEGIN
        Execution statements;
[EXCEPTION]
        Exception handling statements;
END;
```

2. PL/SQL Variables

It shows you how to work with PL/SQL variables including declaring, naming, and assigning variables. In PL/SQL, a variable is a meaningful name of a temporary storage location that supports a particular data type in a program. Before using a variable, you need to declare it first in the declaration section of a PL/SQL block.

PL/SQL variables naming rules

Like other programming languages, a variable in PL/SQL must follow the naming rules as follows:

- The variable name must be less than 31 characters. Try to make it as meaningful as possible within 31 characters.
- The variable name must begin with an ASCII letter. It can be either lowercase or uppercase. Notice that PL/SQL is case-insensitive, which means v_data and V_DATA refer to the same variable.
- Followed by the first character are any number, underscore (_), and dollar sign ($) characters. Once again, do not make your variables hard to read and difficult to understand.

PL/SQL variables naming convention

- It is highly recommended that you should follow the naming conventions listed in the following table to make the variables obvious in PL/SQL programs.

Prefix	Data Type
v_	VARCHAR2
n_	NUMBER
t_	TABLE
r_	ROW
d_	DATE
b_	BOOLEAN

- Each organization has its own development naming convention guidelines. Make sure that you comply with your organization's naming convention guidelines.
- For example, if you want to declare a variable that holds the first name of the employee with the VARCHAR2 data type, the variable name should be v_first_name.

PL/SQL Variables Declaration

- To declare a variable, you use a variable name followed by the data type and terminated by a semicolon (;). You can also explicitly add a length constraint to the data type within parentheses. The following illustrates some examples of declaring variables in a PL/SQL anonymous block:

```
DECLARE
    v_first_name varchar2(20);
    v_last_name varchar2(20);
    n_employee_id number;
    d_hire_date date;
BEGIN
```

```
        NULL;
    END;
```

3. PL/SQL Condition Statements

It introduces you to various forms of the PL/SQL IF statement including IF-THEN, IF-THEN-ELSE and IF-THEN-ELSIF statement.

Conditional statements in PL/SQL are used to execute a block of code only when a certain condition is met. PL/SQL supports various forms of conditional statements, including:

IF-THEN

```
IF condition THEN
   -- code to execute if condition is TRUE
END IF;
```

IF-THEN-ELSE: Provides an alternative block of code to execute when the condition is FALSE.

```
IF condition THEN
   -- code to execute if condition is TRUE
ELSE
   -- code to execute if condition is FALSE
END IF;
```

IF-THEN-ELSIF : Used when you want to test multiple conditions.

```
IF condition1 THEN
   -- code to execute if condition1 is TRUE
ELSIF condition2 THEN
   -- code to execute if condition2 is TRUE
ELSE
   -- code to execute if none of the conditions are TRUE
END IF;
```

4. PL/SQL CASE Statement

It shows you how to use PL/SQL CASE statement and PL/SQL searched CASE statement. The `CASE` statement allows you to select one of several possible actions based on the value of a variable.

```
CASE expression
  WHEN value1 THEN
    -- code to execute if expression equals value1
  WHEN value2 THEN
```

```
    -- code to execute if expression equals value2
  ELSE
    -- code to execute if no matches
END CASE;
```

5. PL/SQL Loop Statement

It guides you on how to use PL/SQL LOOP statement to execute a block of code repeatedly. Loops allow you to repeatedly execute a block of code while a condition holds true. PL/SQL supports different types of loops:

This is an infinite loop unless explicitly terminated using EXIT.

```
LOOP
  -- code to execute repeatedly
EXIT WHEN condition;
  -- exit when condition is TRUE
END LOOP;
```

6. PL/SQL WHILE Loop Statement

It executes a sequence of statements with a condition that is checked at the beginning of each iteration with the WHILE loop statement.

The WHILE loop executes as long as the condition is TRUE.

```
WHILE condition LOOP
  -- code to execute repeatedly
END LOOP;
```

7. PL/SQL FOR Loop Statement

It shows you how to execute a sequence of statements in a fixed number of times with FOR loop statement. The FOR loop iterates over a specified range of values.

```
FOR counter_variable IN lower_bound..upper_bound LOOP
  -- code to execute repeatedly
END LOOP;
```

8. FOR LOOP statement with REVERSE

The FOR loop can also count in reverse using the REVERSE keyword.

```
FOR counter_variable IN REVERSE lower_bound..upper_bound LOOP
  -- code to execute repeatedly
END LOOP;
```

9. **PL/SQL Nested Block** – explains what a PL/SQL nested block is and how to apply it in PL/SQL programming.

10. **PL/SQL Function –** explains what PL/SQL functions are and shows you how to create PL/SQL functions.
11. **PL/SQL Procedure** – discusses PL/SQL procedures and shows you how to create PL/SQL procedures.
12. **PL/SQL Exception Handling** – teaches you how to handle exceptions properly in PL/SQL as well as shows you how to define your own exception and raise it in your code.
13. **PL/SQL Record** – explains the PL/SQL record and shows you how to use records to manage your data more effectively.
14. **PL/SQL Cursor** – covers PL/SQL cursor concept and walks you through how to use a cursor to loop through a set of rows and process each row individually.
15. **PL/SQL Packages** – shows you how to create a PL/SQL package that is a group of related functions, procedures, types, etc.

Implementation of the PL/SQL block

A **PL/SQL block** in Oracle is the basic unit of a PL/SQL program. It consists of three sections: the **declarative**, **executable**, and **exception-handling** sections. Here's a structured guide and an example of implementing a PL/SQL block.

Structure of a PL/SQL Block

1. **Declarative Section**
   - It defines variables, constants, cursors, etc.
   - It starts with `DECLARE` keyword
   - Optional.
2. **Executable Section**
   - It contains the procedural code.
   - It starts with `BEGIN` keyword
   - Mandatory.
3. **Exception-Handling Section**
   - It handles runtime errors.
   - It starts with `EXCEPTION` keyword
   - Optional.
4. **End of the Block**
   - It ends with `END;`

Examples of a PL/SQL Block

1. **Write the simple PL/SQL program to print 'Hello World'**

```
DECLARE
  M varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(M);
```

```
END;
/
```

2. **Write the simple PL/SQL program assigns values from the Customer table to PL/SQL variables using the SELECT INTO clause of SQL.**

```
DECLARE
  c_id customers.id%type := 1;
  c_name  customers.name%type;
  c_addr customers.address%type;
  c_sal  customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal
  FROM customers
  WHERE id = c_id;
  dbms_output.put_line
  ('Customer ' || c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

3. **Write the simple PL/SQL program the given number is less than 20 if not print the given number.**

```
  DECLARE
  A number(3) := 500;
BEGIN
  -- check the boolean condition using if statement
  IF( A < 20 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('A is less than 20 ' );
  ELSE
    dbms_output.put_line('A is not less than 20 ' );
  END IF;
  dbms_output.put_line('value of a is : ' || A);
END;
```

4. **Write the simple PL/SQL program print the message according to the GRADE using CASE.**

```
DECLARE
  grade char(1):='A';
```

```
BEGIN
  CASE grade
    when 'A' then
          dbms_output.put_line('Result is: Excellent');
    when 'B' then
          dbms_output.put_line('Result is: Very good');
    when 'C' then
          dbms_output.put_line('Result is: Good');
    when 'D' then
          dbms_output.put_line('Result is: Average');
    when 'F' then
           dbms_output.put_line('Result is: Passed with Grace');
    else
          dbms_output.put_line('Result is: Failed');
  END CASE;
  End
END;
```

5. **Write the simple PL/SQL program to print the numbers 1-10 using LOOP**

```
DECLARE
i NUMBER := 1;
BEGIN
LOOP
EXIT WHEN i>10;
DBMS_OUTPUT.PUT_LINE(i);
i := i+1;
END LOOP;
END;
```

Procedure and Function in PL/SQL

In PL/SQL, a procedure and a function are both subprograms that encapsulate a block of code for reuse. They share similar structures but differ in their purpose and behavior.

Procedure in PL/SQL

A **procedure** is a subprogram that performs a specific task. It does **not return a value** directly but can return data via **OUT parameters**.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name (parameter_name [IN | OUT | IN OUT] data_type, ...)

IS

   -- Declarations (optional)

BEGIN

   -- Statements

   -- Code to perform the task

END procedure_name;
```

Where,

- **CREATE OR REPLACE:** Optional. Replaces the procedure if it already exists.
- **procedure_name:** The name of the procedure.
- **parameter_name:** The name of a parameter passed to the procedure.
  - ✓ **IN:** Input parameter (default). Used to pass a value to the procedure.
  - ✓ **OUT:** Output parameter. Used to return a value to the caller.
  - ✓ **IN OUT:** Input and output parameter. Used to pass a value to the procedure and return a modified value.

Example:

Procedure to Update Employee Salary

```
CREATE OR REPLACE PROCEDURE UpdateSalary ( emp_id IN NUMBER,

   percent_increase IN NUMBER)

AS

BEGIN

   UPDATE employees

   SET salary = salary + (salary * percent_increase / 100)

   WHERE employee_id = emp_id;

      dbms_output.put_line('Salary updated for Employee ID: ' || emp_id);
```

```
END;
/
```

How to Call the Procedure?

```
BEGIN
    UpdateSalary(A101, 10);  -- Increase salary by 10% for employee with ID A101
END;
```

Function in PL/SQL

A **function** is a subprogram that **returns a single value**. Functions are often used for calculations or retrieving a value.

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name (parameter_name [IN] data_type, ...)
RETURN return_data_type

IS

    -- Declarations (optional)

BEGIN

    -- Statements

    -- Code to perform the task

    RETURN value;  -- Return a single value

END function_name;
```

Where,

- **CREATE OR REPLACE**: Optional. Replaces the function if it already exists.
- **function name:** The name of the function.
- **parameter_name:** The name of the input parameter.
- Note that Functions only allow **IN** parameters.
- **RETURN return_data_type**: Specifies the data type of the value returned by the function.
- **IS/AS**: Begins the declaration section.
- **BEGIN**: Begins the executable section.
- **RETURN**: Returns a value to the calling program.
- **END:** Ends the function.

Example: Function to Calculate Total Salary

```
CREATE OR REPLACE FUNCTION GetTotalSalary ( emp_id IN varchar )
RETURN NUMBER
AS
   total_salary NUMBER;
BEGIN
   SELECT SUM(salary)  INTO total_salary  FROM employee WHERE emp_id = emp_id;
   RETURN total_salary;
END;
/
```

How to Call the Function?

```
DECLARE
   total NUMBER;
BEGIN
   total := GetTotalSalary(10);
   dbms_output.put_line('Total Salary: ' || total);
END;
/
```