UNIT 5:
Oracle Database Objects

Installing PL/SQL Packages in Oracle Database

1. Download the Oracle Graph PL/SQL patch component, which is a part of the Oracle Graph Server and Client download from Oracle Software Delivery Cloud.
2. Unzip the file oracle-graph-plsql-<*ver*>.zip into a directory of your choice. <*ver*> denotes the version downloaded for the Oracle Graph PL/SQL Patch for PL/SQL.
3. Install the PL/SQL packages:
    a. There are two directories, one for users with Oracle Database 18c or below, and one for users with Oracle Database 19c or above. As a database user with DBA privilges, follow the instructions in the README.md file in the appropriate directory (that matches your database version). This has to be done for every PDB you will use the graph feature in. For example:

    ```
    Copy
    -- Connect as SYSDBA
    SQL> ALTER SESSION SET CONTAINER=<YOUR_PDB_NAME>;
    SQL> @opgremov.sql
    SQL> @catopg.sql
    ```
4. Create a database user in the database for working with graphs:
    a. As a database user with DBA privileges, create a user <graphuser>, and grant the necessary privileges.
        i. If you plan to use a three-tier architecture (graph queries and analytics executed in the in-memory graph server (PGX), then grant privileges as described in the following command:

        ```
        Copy
        ```

```
SQL> GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW TO
<graphuser>
```

ii. If you plan to use a two-tier architecture and run graph queries in the database, then grant privileges as described in Required Privileges for Database Users:

```
Copy
SQL> GRANT CREATE SESSION, ALTER SESSION, CREATE TABLE, CREATE
PROCEDURE, CREATE TYPE, CREATE SEQUENCE, CREATE VIEW, CREATE
TRIGGER TO <graphuser>
```

b. As a <graphuser> in the database, check that the PL/SQL update is successful:

```
Copy
SQL> CONNECT <graphuser>/<password>
SQL> SELECT opg_apis.get_opg_version() FROM DUAL;
   -- Should return 21.2 if you are using
   -- Graph Server and Client 21.2
```

5. Grant the appropriate roles (GRAPH_DEVELOPER or GRAPH_ADMINISTRATOR), to the database user created in step 4 for working with the graphs.

   Note:

   - See User Authentication and Authorization for more information on authorization rules for Graph Server (PGX) and Client 21.2.
   - See Upgrading From Graph Server and Client 20.4.x to 21.x for more information if you are migrating to Graph Server (PGX) and Client 21.1 from an earlier version.

```
Copy

SQL> GRANT GRAPH_DEVELOPER to <graphuser>
SQL> GRANT GRAPH_ADMINISTRATOR to <adminuser>
```

PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.

- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

> ***A procedure may or may not return any value.***

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

```
1. create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

Procedure Code:

```
1. create or replace procedure "INSERTUSER"
2. (id IN NUMBER,
3. name IN VARCHAR2)
4. is
5. begin
6. insert into user values(id,name);
7. end;
8. /
```

Output:

```
Procedure created.
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
1. BEGIN
2.    insertuser(101,'Rahul');
3.    dbms_output.put_line('record inserted successfully');
4. END;
5. /
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

PL/SQL Drop Procedure

Syntax for drop procedure

```
DROP PROCEDURE procedure_name;
```

Example of drop procedure

```
DROP PROCEDURE pro1;
```

PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Here:

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.

- The AS keyword is used instead of the IS keyword for creating a standalone function.

PL/SQL Function Example

Let's see a simple example to **create a function**.

```
1. create or replace function adder(n1 in number, n2 in number)
2. return number
3. is
4. n3 number(8);
5. begin
6. n3 :=n1+n2;
7. return n3;
8. end;
9. /
```

Now write another program to **call the function**.

```
1. DECLARE
2.    n3 number(2);
3. BEGIN
4.    n3 := adder(11,22);
5.    dbms_output.put_line('Addition is: ' || n3);
6. END;
7. /
```

Output:

```
Addition is: 33
Statement processed.
0.05 seconds
```

Another PL/SQL Function Example

Let's take an example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

```
1. DECLARE
2.    a number;
```

```
3.    b number;
4.    c number;
5.  FUNCTION findMax(x IN number, y IN number)
6.  RETURN number
7.  IS
8.     z number;
9.  BEGIN
10.   IF x > y THEN
11.     z:= x;
12.   ELSE
13.     Z:= y;
14.   END IF;
15.
16.   RETURN z;
17. END;
18. BEGIN
19.   a:= 23;
20.   b:= 45;
21.
22.   c := findMax(a, b);
23.   dbms_output.put_line(' Maximum of (23,45): ' || c);
24. END;
25. /
```

Output:

```
Maximum of (23,45): 45
Statement processed.
0.02 seconds
```

PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

Create customers table and have records in it.

Customers		
Id	**Name**	**Department**
1	alex	web developer
2	ricky	program developer
3	mohan	web designer
4	dilshad	database manager

Create Function:

```
1. CREATE OR REPLACE FUNCTION totalCustomers
2. RETURN number IS
3.   total number(2) := 0;
4. BEGIN
5.   SELECT count(*) into total
6.   FROM customers;
7.    RETURN total;
8. END;
9. /
```

After the execution of above code, you will get the following result.

```
Function created.
```

Calling PL/SQL Function:

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

After the execution of above code in SQL prompt, you will get the following result.

```
Total no. of Customers: 4
PL/SQL procedure successfully completed.
```

PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
```

```
10.    IF x=0 THEN
11.       f := 1;
12.    ELSE
13.       f := x * fact(x-1);
14.    END IF;
15. RETURN f;
16. END;
17.
18. BEGIN
19.    num:= 6;
20.    factorial := fact(num);
21.    dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
22. END;
23. /
```

After the execution of above code at SQL prompt, it produces the following result.

```
Factorial 6 is 720
PL/SQL procedure successfully completed.
```

PL/SQL Drop Function

Syntax for removing your created function:

If you want to remove your created function from the database, you should use the following syntax.

```
1.  DROP FUNCTION function_name;
```

PL/SQL Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating a trigger:

Syntax for creating trigger:

1. **CREATE** [OR REPLACE] **TRIGGER** trigger_name
2. {BEFORE | **AFTER** | **INSTEAD OF** }
3. {**INSERT** [OR] | **UPDATE** [OR] | **DELETE**}
4. [**OF** col_name]
5. **ON** table_name
6. [REFERENCING OLD **AS** o NEW **AS** n]
7. [**FOR** EACH ROW]
8. **WHEN** (condition)

9. **DECLARE**
10. Declaration-statements
11. **BEGIN**
12. Executable-statements
13. EXCEPTION
14. Exception-handling-statements
15. **END**;

Here,

- CREATE [OR REPLACE] TRIGGER trigger_name: It creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

Create table and have records:

ID	NAME	AGE	ADDRESS

1	Ramesh	23	Allahabad
2	Suresh	22	Kanpur
3	Mahesh	24	Ghaziabad
4	Chandan	25	Noida
5	Alex	21	Paris
6	Sunita	20	Delhi

Create trigger:

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
1.  CREATE OR REPLACE TRIGGER display_salary_changes
2.  BEFORE DELETE OR INSERT OR UPDATE ON customers
3.  FOR EACH ROW
4.  WHEN (NEW.ID > 0)
5.  DECLARE
6.    sal_diff number;
7.  BEGIN
8.    sal_diff := :NEW.salary  - :OLD.salary;
9.    dbms_output.put_line('Old salary: ' || :OLD.salary);
10.   dbms_output.put_line('New salary: ' || :NEW.salary);
11.   dbms_output.put_line('Salary difference: ' || sal_diff);
12. END;
13. /
```

After the execution of the above code at SQL Prompt, it produces the following result.

```
Trigger created.
```