PL SQL

- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.
- PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

Sections & Description

- **Declarations**
- This section starts with the keyword **DECLARE**
- **Executable Commands**
- This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section.
- It consists of the executable PL/SQL statements of the program.
- It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
- Every PL/SQL statement ends with a semicolon (;).

Basic structure of a PL/SQL block

- DECLARE
- <declarations section>
- BEGIN
- <executable command(s)>
- EXCEPTION
- <exception handling>
- END;

The 'Hello World' Example

```
DECLARE
   message varchar2(20): = 'Hello, World!';
BEGIN
   dbms_output.put_line(message);   //display message
END;
Set Serveroutput ON      // To display messages to the user
/
```

PL/SQL Input Output Statements

- **Input Statement**
- There is no input statement or input function to print the values at run time but if you want to input the value at run time then you use insertion operator **(&)**.
- message : = &message;
- radius := &radius;

- **Output Statement**
- Output statements are used for print output on console, pl/sql have own output statements to print value on screen which is;

Output Statement

- DBMS_OUTPUT→This package enables display messages on the screen.
-
- PUT_LINE→ We can place an entire line of information into buffer by calling put_line.
- **Example**
- dbms_output.put_line('Hello word');
- dbms_output.put_line(‘sum’IIsum);

Example

- SET SERVEROUTPUT ON→To display messages to the user the server output should be ON.
- Two ways:(Windows) either open a notepad using the command
- ed filename type the code and save it either filename like “pgm1” and copy the contens minimize the notepad screen not closed and then paste the contents into prompt. Then it will run (/ and ‘set serverputput on’ command is necessary).Otherwise write the code in command prompt directly.

- Linux: write the code in command prompt directly.

- For run the command two ways:

- **1, use / symbol in the code ……..(first method)**
- **2, Save file-> save file name.sql …….(second method)**
- **Execution ->@filename.**
-

PL/SQL Loop

- for variable in initial value..final value
- Loop
- Statements
- End loop;

Example: Print **Hello** word five times

- Declare
- i number(3);
- Begin
- For i in 1..4
- loop
- Dbms_output.put_line(“hello”)
- End loop;
- End;

Eg: Find area of the circle:

- DECLARE
- area   number(6, 2) ;
- pi constant number(3, 2) := 3.14;
- radius number(5) := 3;

- Begin
- area := pi * radius * radius;

- dbms_output.Put_line('Area = ' || area);
. end;

```
DECLARE
area   number(6, 2)  ;
pi constant number(3, 2) := 3.14;
radius number(5) ;
 Begin
radius := &radius;
area := pi * power(radius,2)
  dbms_output.Put_line(‘radius  is= ' || radius);
  dbms_output.Put_line('Area = ' || area);
.  end;
Set Serveroutput ON
/
```

Branching

- If (condition) then
- Statement;
- Else if (condition) then
- Statement;
- Else
- Statement;
- End if;

Eg:

```
Declare
n integer :=&n;
Begin
  if(n>10) then
Dbms_output.put_line(n)
End if;
End;
```

Loop reverse:

- For variable in(reverse) lower bound..upper bound
- Loop
- Sequence of statements
- End loop;

Eg: Program to inverting a number

- Declare
- i number(5);
- Str varchar(10);
- Rev varchar(10);
- Len number(2);
- Begin
- Str := &str;
- Len := length(str)
- For i in reverse 1..Len
- Loop
- Rev := Rev II SUBSTR(Str,i, 1);
- End loop;
- Dbms_output.Put_line(‘given no’ II str);
- Dbms_output.Put_line(‘Reverse’ IRev);
- End;

Program list for next lab

- **Prog 9: Write a PL/SQL program to calculate area of a circle.**
-
- **Prog 10: Execute QN no 9 using LOOP and Branch Statements.**
-
- **Prog 11: Write a PL/SQL program to find factorial of a number.**
-
- **Prog 12: Write a PL/SQL program to check whether a number is odd or even.**
- **Prog 13: Write a PL SQL code for reverse a string.**

PL SQL Functions

- A function is logically grouped set of SQL & PL SQL statements that performs a specific task.
- Functions are made up of
- Declarative part(it may contain declaration of constants, variables etc.)
- Executable part(It consist SQL and PL/SQL statements that assign values, control execution and manipulation of data)
- PL SQL function must return value.

Syntax

- Create or Replace Function<Function name>

  (<argument> in <Data type>, …….)

Return < Data type > { is ,as }

  <variable> declarations;

Begin

  <PL/SQL subprogram body>

End function name;

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The function must contain a **return** statement.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Eg1:PL/SQL Function that computes and returns the maximum of two values.

```
Create or Replace function findMax(x IN number, y IN number)
RETURN number IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;
  RETURN z;
END findMax;
```

Function Call

- **DECLARE**
- **a number;**
- **b number;**
- **c number;**
- **BEGIN**
- **a:= 23;**
- **b:= 45;**
- **c := findMax(a, b);**
- **dbms_output.put_line(' Maximum of (23,45): ' || c);**
- **END;**
- **/**
- **When the above code is executed at the SQL prompt, it produces the following result –**
- **Maximum of (23,45): 45**
- **PL/SQL procedure successfully completed.**

Eg:2-Function : Add two numbers

- Set Serveroutput ON
- Create or replace function addnum (val1 in number,val2 in number)
- return number is
- Total number;
- Begin
- Total := val1+val2;
- Return(total);
- End addnum;
- After compiling this function we can use this in our SQL query
- Select addnum(3,4) from dual;
- O/P 7

Eg:3-Create a table squares to store a set of values and their corresponding square values

```
Set Serveroutput ON
Create or replace function findsquare (num in number)
return number is
sq number(10);
Begin
Sq:= num*num;
Return sq;
End findsquare;
/
```

Function call

- Declare
  i number(3);
  sq number(10);

```
Begin
   for i in 1..10
   loop
  sq := findsquare(i);
Insert into squares values(i,sq);
End loop
End;
/
Select * from squares;
```

PL-SQL PROCEDURE

- Procedure can accept multiple input parameters and return multiple output parameters.
- In function can return a value but in procedures it is optional. It can return zero or n values.
- By defining multiple out parameters in procedures, multiple values can be passed to the caller.

Syntax

- Create or Replace Procedure<Procedure name>

  (argument {in, out, inout} , Data type, …….)

  { is ,as }

  <variable> declarations;

Begin

  <PL/SQL subprogram body>

End procedure name;

Parameter Mode & Description

- IN – Indicating the parameter will accept a value from the user or subprogram or calling program.
- It is a read only parameter, it is the default mode of parameter passing.
- OUT – Indicate the parameter will return a value to the user.
- IN OUT – Indicates the parameter will either accept a value from the user or return a value to the user.

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

- Set Serveroutput ON;
- Create or Replace PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
- BEGIN
- IF x < y THEN
- z:= x;
- ELSE
- z:= y;
- END IF;
- END;

Calling Program

```
DECLARE
   a number;
   b number;
   c number;
BEGIN
   a:= &a;
   b:= &b;
findMin(a, b, c);
   dbms_output.put_line(' Minimum value is: ' || c);
END;
/
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return the result.

- Set Serveroutput ON;
- Create or Replace PROCEDURE squareNum(x IN OUT number) IS
- BEGIN
- x := x * x;
- END;
- /

Calling Program

```
DECLARE
   a number;
BEGIN
   a:= &a;
   squareNum(a);
   dbms_output.put_line(' Square is): ' || a);
END;
/
```

Eg.3 A Procedure called Deposit is created and stored in database. Create the table customer(A/c no, balance) and update the balance using the procedure Deposit.

- Create table customer(A/c no,balance)
- Insert some values
- Select * from customer
- Set serveroutput ON
- Create or replace procedure deposit(id in number, amt in number)is
- Begin
- Update customer set balance=balance + amt where A/c no=id;
- End;

Program to deposit an amount in customer account

- Declare
- Accno number(2);
- Amount number(10,2);
- Begin
- Accno:= &Accno;
- Amount:=&amount;
- Deposit(Acc no, amount);
- Commit;
- End;

PL/SQL - Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur.
- Triggers can be defined on the table, schema, or database with which the event is associated.
- Triggers are, in fact, 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).

Benefits of Triggers

- Enforcing complex integrity constraints.
- validating input data.
- Imposing security authorizations.
- Preventing invalid transactions.

Parts of trigger

- Event : That activates the trigger.
- Condition : Test whether the trigger should run.(optional)
- Action : What happens if the trigger runs.

- CREATE [OR REPLACE] TRIGGER trigger_name
- {BEFORE | AFTER }
- {INSERT [OR] | UPDATE [OR] | DELETE}
- [OF col_name]
- ON table_name
- [REFERENCING OLD AS o NEW AS n]
- [FOR EACH ROW]
- WHEN (condition)
- DECLARE
- Declaration-statements
- BEGIN
- Executable-statements
- EXCEPTION
- Exception-handling-statements
- END;

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER } − This specifies when the trigger will be executed.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will 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, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will 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.

Eg: statement level trigger:

- Statement level triggers executes only once for each single transaction.
- Create table xy(id, name, age);
- Insert three values;
- Display table xy;
- Create table testtable(action, date);((data type varchar(50) give large value))
- Execute statement level trigger with name t2;

```
Create trigger t2 after insert or update on xy
Begin
If inserting
    then insert into testtable values('insert done', SYSDATE);
  Else
    insert into testtable values('update done', SYSDATE);
End if;
End;
/
>trigger created.
```

- This is statement level trigger. That is if we update all rows in xy, trigger fire only one time.
- Select * from testtable;
- No rows selected.
- Insert two more values into xy table;
- Select * from testtable;
- Update name on xy table belongs to either id;
- Select * from testtable;

ROW level trigger

- Row level triggers executes once for each and every row in the transaction.
- The following program creates 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 −

Eg:2, ROW level trigger: CUSTOMERS table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
    WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```