TRIGGERS

Trigger defines an action the database should take when some database-related event occurs. Triggers are executed by the database when specific types of data manipulation commands are performed on specific tables. Such commands may include **insert**s, **update**s, and **delete**s. Updates of specific columns may also be used as triggering events.

Types of Triggers

A trigger's type is defined by the type of triggering transaction and by the level at which the trigger is executed.

Row-Level Triggers

Row-level triggers execute once for each row in a transaction.
Row-level triggers are created using the **for each row** clause in the **create trigger** command.

Statement-Level Triggers

Statement-level triggers execute once for each transaction. Statement-level triggers are the default type of trigger created via the **create trigger** command.

The general **syntax** of the trigger is:

Create [or replace] trigger Trigger_name
[Before / after] [Insert/update/delete]
On Table_name
[For each row]
[When condition]
Declare
Declaration statements
Begin
Executable statements
Exception
Exception handling statements
End;

The **before** and **after** keywords indicate whether the trigger should be executed before or after the triggering transaction.

The **delete**, **insert**, and **update** keywords (the last of which may include a column list) indicate the type of data manipulation that will constitute a triggering event. When referring to the old and new values of columns, you can use the defaults ("old" and "new") or you can use the **referencing** clause to specify other names.

When the **for each row** clause is used, the trigger will be a row-level trigger; otherwise, it will be a statement-level trigger.

The **when** clause is used to further restrict when the trigger is executed. The restrictions enforced in the **when** clause may include checks of old and new data values.

Example:

```
Create or replace trigger trig
Before insert on loan
For each row
Begin
If inserting then
Dbms_output.put_line(‘value inserted’);
End  if;
End;
```

```
SQL> insert into employee values('john',122,2000);
value insetred
1 row created.
```

PL/SQL

(PROCEDURAL LANGUAGE/ STRUCTURED QUERY LANGUAGE)

PL/SQL stands for Procedural Language/SQL. PL/SQL is super set of the Structured Query Language. Using PL/SQL user can do things like codify business rules through the creation of stored procedures and packages, trigger database events to occur, or add programming logic to the execution of SQL commands.

PL/SQL code is grouped into structures called blocks.

Anonymous block - If the block of PL/SQL code is not given a name then it is called anonymous block.

Named block - block with a code with a name. Procedures and Functions are examples.

Features of PL/SQL

- Block(modular) structures
- Flow-control statements and loops
- Variables, constants and types
- Structured data
- Customized error handling
- Allows to store compiled code directly in the database

PL/SQL datatypes include all of the valid SQL datatypes as well as complex datatypes based on query structures.

A block of PL/SQL contains three sections (for anonymous PL/SQL block)

Section	**Description**
Declarations	Defines and initializes the variable and cursors used in the block
Executable commands	Uses flow-control commands (such as if commands and loops) to execute the commands and assign values to the declared variables
Exception Handling	Provides customized handling of error conditions

Structure of a PL/SQL block

declare
<declarations section>
begin
<executable commands>

```
exception
<exception handling>
end;
```

Declarations Section

The Declarations section begins a PL/SQL block. The Declarations section starts with the declare keyword, followed by a list of variable and cursor definitions.

User can define variables to have constant values, and variables can inherit datatypes from existing columns and query results.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
radius INTEGER(5);
area NUMBER(14,2);
begin
<executable commands>
end;
/
```

%TYPE and %ROWTYPE

Both %TYPE and %ROWTYPE are used to define variables in PL/SQL as it is defined within the database. If the datatype or precision of a column changes, the program automically picks up the new definition from the database without having to make any code changes. The %TYPE and %ROWTYPE constructs allows programs to adapt as the database changes to meet new business needs.

%TYPE

%TYPE is used to declare a field with the same type as that of a specified table's column.

Example:

```
DECLARE
  v_EmpName  emp.ename%TYPE;
BEGIN
  SELECT ename INTO v_EmpName FROM emp WHERE ROWNUM = 1;
  DBMS_OUTPUT.PUT_LINE('Name = ' || v_EmpName);
END;
```

/

%ROWTYPE

%ROWTYPE is used to declare a record with the same types as found in the specified database table, view or cursor. Examples:

```
DECLARE
  v_emp emp%ROWTYPE;
BEGIN
  v_emp.empno := 10;
  v_emp.ename := 'JOY';
END;
/
```

Executable Commands Section

In the Executable Commands section, user can manipulate the variables and cursors declared in the Declarations section of your PL/SQL block. The Executable Commands section always starts with the keyword begin.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
radius INTEGER(5);
area NUMBER(14,2);
begin
radius := 3;
area := pi*power(radius,2);
insert into AREAS values (radius, area);
end;
/
```

Exception Handling Section

When user-defined or system-related exceptions (errors) are encountered, the control of the PL/SQL block shifts to the Exception Handling section. Within the Exception Handling section, the when clause is used to evaluate which exception is to be “raised”—that is, executed.

If an exception is raised within the Executable Commands section of your PL/SQL block, the flow of commands immediately leaves the Executable Commands section and searches the Exception Handling section for an exception matching the error encountered. PL/SQL provides a set of system-defined exceptions and allows you to add your own exceptions.

Every exception in PL/SQL has an error number and error message; some exceptions also have names.

Predefined Exception

Syntax:

Begin
Sequence of statements;
Exception
When <exception_name> then
Sequence of statements;
End
Declaring Exceptions

```
declare
mark student.mark1%type;
begin
select mark1 into mark from student where mark1>95;
dbms_output.put_line('current'    :  || mark );
exception
when no_data_found then
dbms_output.put_line('no such student');
end;
/
```

Raising Exceptions

An exception can be raised in three ways:

- By the PL/SQL runtime engine
- By an explicit RAISE statement in your code
- By a call to the built-in function RAISE_APPLICATION_ERROR

The **syntax** for the RAISE statement is:

RAISE exception_name;

```
declare
 lo_mark exception;
 mark student.mark1%type;
 begin
 select mark1 into mark from student where sname='johny';
 if mark < 50 then
```

```
raise lo_mark;
end if;
exception
when lo_mark then
dbms_output.put_line('student has got less mark');
end;
/
```

Example:

Raise_application_error

It is used to create user defined error messages.

Syntax:

raise_application_error(error number, error message);

where , error number – between –20,000 and –20,999.
Error message – text associated with this error.

Example:

```
exception lo_bal then
raise_application_error(-20001, ‘ balance is low’);
--
end;
```

Conditional Logic

Within PL/SQL, user can use if, else, and elsif commands to control the flow of commands within the Executable Commands section

Syntax:

```
if <some condition>
then <some command>
elsif <some condition>
then <some command>
else <some command>
end if;
```

Example:

```
if area >30
then
insert into AREAS values (rad_val.radius, area);
end if;
```

Loops

User can use loops to process multiple records within a single PL/SQL block. PL/SQL supports three types of loops

Simple loops

A loop that keeps repeating until an exit or exit when statement is reached within the loop. The loop is started by the loop keyword, and the exit when clause determines when the loop should be exited. An end loop clause signals the end of the loop.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
radius INTEGER(5);
area NUMBER(14,2);
begin
radius := 3;
loop
area := pi*power(radius,2);
insert into AREAS values (radius, area);
radius := radius+1;
exit when area >100;
end loop;
end;
/
```

FOR Loops

A loop that repeats a specified number of times. The FOR loop's start is indicated by the keyword **for**, followed by the criteria used to determine when the processing should exit the loop. Since the number of times the loop is executed is set when the loop is begun, an **exit** command isn't needed within the loop.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
radius INTEGER(5);
area NUMBER(14,2);
begin
for radius in 1..7 loop
area := pi*power(radius,2);
insert into AREAS values (radius, area);
end loop;
end;
```

/

WHILE Loops

In a WHILE loop, the loop is processed until an exit condition is met. Instead of specifying the exit condition via an **exit** command within the loop, the exit condition is specified in the **while** command that initiates the loop.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
radius INTEGER(5);
area NUMBER(14,2);
begin
radius := 3;
while radius<=7
loop
area := pi*power(radius,2);
insert into AREAS values (radius, area);
radius := radius+1;
end loop;
end;
/
```

Cursor

A cursor is handle or pointer to the context area. The PL/SQL program can control the context area by using the cursor. There can be either static cursors, whose SQL statement is determined at compile time, or dynamic cursors, whose SQL statement is determined at runtime.

Types of Cursors

PL/SQL uses two types of cursors***:*** *explicit* **and** *implicit*

Explicit Cursors

Explicit cursors are SELECT statements that are DECLAREd explicitly in the declaration section of the current block or in a package specification. Use OPEN, FETCH, and CLOSE in the execution or exception sections of your programs.

Declaring explicit cursors

To use an explicit cursor, first declare it in the declaration section of a block or package. There are three types of explicit cursor declarations:

A cursor without parameters

cursor company_cur is select company_id from company;

A cursor that accepts arguments through a parameter list

cursor company_cur (id_in in number) is select name from company where company_id = id_in;

A cursor **header** that contains a RETURN clause in place of the SELECT statement:

cursor company_cur (id_in in number) return company%rowtype is select * from company;

Opening explicit cursors

To open a cursor, use the following **syntax:**

OPEN cursor_name [(argument [,argument ...])];

where **cursor_name** is the name of the cursor as declared in the declaration section. The arguments are required if the definition of the cursor contains a parameter list.

```
DECLARE
  CURSOR c1 IS SELECT ename, job FROM employee WHERE sal < 7000;
  ...
BEGIN
  OPEN c1;
  ...
END;
```

Fetching from explicit cursors

The FETCH statement places the contents of the current row into local variables. To retrieve all rows in a result set, each row needs to be fetched.

The **syntax** for a FETCH statement is:

FETCH cursor_name INTO record_or_variable_list;

where **cursor_name** is the name of the cursor as declared and opened. Closing explicit cursors

The **syntax** of the CLOSE statement is:

CLOSE cursor_name;

where **cursor_name** is the name of the cursor declared and opened.

Explicit cursor attributes

Attribute	Description
%ISOPEN	TRUE if cursor is open. FALSE if cursor is not open.
%FOUND	Returns true if the last fetch returned a row else a false
%NOTFOUND	Returns true if the last fetch did not return a row else a false
%ROWCOUNT	The number of rows fetched from the cursor. INVALID_CURSOR if cursor has been CLOSEd.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
area NUMBER(14,2);
cursor rad_cursor is
select * from RADIUS_VALS;
rad_val rad_cursor%ROWTYPE;
begin
open rad_cursor;
loop
fetch rad_cursor into rad_val;
exit when rad_cursor%NOTFOUND;
area := pi*power(rad_val.radius,2);
insert into AREAS values (rad_val.radius, area);
end loop;
close rad_cursor;
end;
/
```

Implicit Cursors

Whenever a SQL statement is directly in the execution or exception section of a PL/SQL block, you are working with implicit cursors. These statements include INSERT, UPDATE, DELETE, and SELECT INTO statements. Unlike explicit cursors, implicit cursors do not need to be declared, OPENed, FETCHed, or CLOSEd.

SELECT statements handle the %FOUND and %NOTFOUND attributes differently from explicit cursors. When an implicit SELECT statement does not return any rows, PL/SQL immediately raises the NO_DATA_FOUND exception and control passes to the exception section. When an

implicit SELECT returns more than one row, PL/SQL immediately raises the TOO_MANY_ROWS exception and control passes to the exception section.

Implicit cursor attributes

Attribute	Description
%ISOPEN	Always false as it is closed immediately after executing its associated SQL statement
%FOUND	Returns true if the last DML statement returned a row else a false
%NOTFOUND	Returns true if the last DML statement did not return any row else a false
%ROWCOUNT	Returns the total number of rows returned.

Cursor FOR loop

In a Cursor FOR loop, the results of a query are used to dynamically determine the number of times the loop is executed. In a Cursor FOR loop, the opening, fetching, and closing of cursors is performed implicitly.

Example:

```
declare
pi constant NUMBER(9,7) := 3.1415926;
area NUMBER(14,2);
cursor rad_cursor is
select * from RADIUS_VALS;
begin
for rad_val in rad_cursor
loop
area := pi*power(rad_val.radius,2);
insert into AREAS values (rad_val.radius, area);
end loop;
end;
/
```

Ref Cursors

To create cursor variables, you take two steps. First, you define a `REF CURSOR` type, then declare cursor variables of that type. You can define `REF CURSOR` types in any PL/SQL block, subprogram, or package using the syntax

TYPE ref_type_name IS REF CURSOR RETURN return_type;

Declaring Cursor Variables

Once you define a `REF CURSOR` type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable `dept_cv`:

```
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
  dept_cv DeptCurTyp;  -- declare cursor variable
```

Opening a Cursor Variable

The `OPEN-FOR` statement associates a cursor variable with a multi-row query, executes the query, and identifies the result set. Here is the syntax:

```
OPEN {cursor_variable_name | :host_cursor_variable_name}
  FOR select_statement;
```

where `host_cursor_variable_name` identifies a cursor variable declared in a PL/SQL host environment such as an OCI or Pro*C program.

Closing a Cursor Variable

The `CLOSE` statement disables a cursor variable. After that, the associated result set is undefined. Here is the syntax:

```
CLOSE {cursor_variable_name | :host_cursor_variable_name);
```

Varying Arrays

A varying array allows you to store repeating attributes of a record in a single row.

Creating a Varying Array

```
SQL> create or replace type mark as varray(5) of number;
      /
```

Type created.

```
SQL> create table varr(no number,marks mark);
```

Table created.

```
SQL> desc varr;
 Name           Null?            Type
 -------------- -------- -------------------------
 NO                              NUMBER
 MARKS                           MARK
```

```
SQL> select * from varr;

        NO                MARKS
----------   --------------------------------

       101      MARK(33, 44, 55, 66)
```

Example:

```
declare
cursor borrower_cursor is
select * from varr;
begin
for borrower_rec in borrower_cursor
loop
for i in 1..borrower_rec.marks.count
loop
dbms_output.put_line(borrower_rec.marks(i));
end loop;
end loop;
end;
/
33
44
55
66
```

PL/SQL procedure successfully completed.

Named Program Units

The PL/SQL programming language allows you to create a variety of named program units they include:

Procedure - A program that executes one or more statements

Function - A program that returns a value

Package - A container for procedures, functions, and data structures

Triggers - Programs that execute in response to database changes

PROCEDURES

Procedures are program units that execute one or more statements and can receive or return zero or more values through their parameter lists.

The syntax of a procedure is:

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

Example:

```
create procedure NEW_WORKER (Person_Name varchar2)
AS
BEGIN
insert into WORKER
(Name, Age, Lodging)
values
(Person_Name, null, null);
END;
/
```

Parameter modes

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

IN	Used to pass values to the subprogram when invoked. It acts like a constant and cannot be assigned a value.
OUT	Used to return values to the caller of a subprogram. It can be assigned some values.
IN OUT	Used to pass initial values to the subprogram when invoked and it also returns updated values to the caller. Can be assigned to other variables or to itself

Calling a Procedure

A call to the procedure is made through an executable PL/SQL statement. The procedure has the following syntax:

Procedure_Name [(parameters)];

The procedure can be executed from SQL environment with the execute command as follows:

EXECUTE Procedure_Name [(parameters)];

Dropping procedure

To drop a procedure, use the drop procedure command, as follows:

drop procedure NEW_WORKER;

FUNCTIONS

Unlike procedures, functions can return a value to the caller (procedures cannot return values). This value is returned through the use of the return keyword within the function.

A function is characterized as follows:

• A function can be with one, more or no parameters.

• A function must have an explicit RETURN statement in the executable section to return a value.

• The data type of the return value must be declared in the function's header.

• A function cannot be executed as a standalone program.

The **syntax** for a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (parameter [,parameter]) ]
   RETURN return_datatype
IS | AS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [function_name];
```

Example:

```
create or replace function sqrtno(no number)
 return number
 is
 begin
 return no*no;
 end;
  /
Function created.
SQL> select sqrtno(5) from dual;
 SQRTNO(5)
----------
        25
```

Dropping function

To drop a function, use the drop function command, as follows:

```
drop function BALANCE_CHECK;
```