



School of Computing Science and Engineering

LAB - 13 Exercises

Course Code	:	CSE2004 - Data Base Management Lab	Date	:	26/10/2018
Lab Experiment	:	PL SQL Triggers	Slots	:	L29+L30
Instructors	:	Prof. Ramesh Ragala			

PL/SQL Triggers

AIM:

To develop and execute a Trigger for Before and After update, delete, insert operations on a table.

PROCEDURE:

STEP 1: Initialize the trigger with specific table id.

STEP 2: Specify the operations (update, delete, insert) for which the trigger has to be executed.

STEP 3: Execute the Trigger procedure for both before and after sequences

STEP 4: Carryout the operation on the table to check for Trigger execution.

TRIGGERS:

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed. Sometimes a trigger is referred as a special kind of stored procedure in term of procedural code



inside its body. The difference between a trigger and a stored procedure is that a trigger is activated or called when an event happens in a database table, a stored procedure must be called explicitly. For example you can have some business logic to do before or after inserting a new record in a database table.

Types of PL/SQL Triggers:

There are two types of triggers based on which level it is triggered.

- 1) Row level trigger** - An event is triggered for each row updated, inserted or deleted.
- 2) Statement level trigger** - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy:

The following hierarchy is followed when a trigger is fired.

- 1) BEFORE** statement trigger fires first.
- 2) Next BEFORE** row level trigger fires, once for each row affected.
- 3) Then AFTER** row level trigger fires once for each affected row. These events will alternates between BEFORE and AFTER row level triggers.
- 4) Finally the AFTER** statement level trigger fires.

The Syntax for creating a trigger is:

```
CREATE [OR REPLACE] TRIGGER trigger_name  
  
    { BEFORE | AFTER | INSTEAD OF }  
  
    { INSERT [OR] | UPDATE [OR] | DELETE }  
  
    [OF col_name]  
  
    ON table_name  
  
    [REFERENCING OLD AS o NEW AS n]  
  
    [FOR EACH ROW]
```



WHEN (condition)

BEGIN

--- sql statements

END;

- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE / AFTER / INSTEAD OF }* - This clause indicates at what time the trigger should get fired. i.e. for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] / UPDATE [OR] / DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot



reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Example:

The price of a product changes constantly. It is important to maintain the history of the prices of the products. We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

- 1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
```

```
(product_id number(5),  
product_name varchar2(32),  
supplier_name varchar2(32),  
unit_price number(7,2) );
```

```
CREATE TABLE product
```

```
(product_id number(5),  
product_name varchar2(32),  
supplier_name varchar2(32),  
unit_price number(7,2) );
```



- 2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger  
  
BEFORE UPDATE OF unit_price  
  
ON product  
  
FOR EACH ROW  
  
BEGIN  
  
INSERT INTO product_price_history  
  
VALUES  
  
(:old.product_id,  
  
:old.product_name,  
  
:old.supplier_name,  
  
:old.unit_price);  
  
END;  
  
/
```

- 3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

- 4) If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.



Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product  
  
(Message varchar2 (50),  
  
Current Date number(32)  
  
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product  
  
BEFORE  
  
UPDATE ON product  
  
Begin  
  
INSERT INTO product_check  
  
Values('Before update, statement level',sysdate);  
  
END;  
  
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
CREATE or REPLACE TRIGGER Before_Upddate_Row_product
```



BEFORE

UPDATE ON product

FOR EACH ROW

BEGIN

INSERT INTO product_check

Values('Before update row level',sysdate);

END;

/

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

CREATE or REPLACE TRIGGER After_Update_Stat_product

AFTER

UPDATE ON product

BEGIN

INSERT INTO product_check

Values('After update, statement level', sysdate);

End;

/

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

CREATE or REPLACE TRIGGER After_Update_Row_product



AFTER

insert On product

FOR EACH ROW

BEGIN

INSERT INTO product_check

Values('After update, Row level',sysdate);

END;

/

Now let's execute an update statement on table product.

UPDATE PRODUCT SET unit_price = 800

WHERE product_id in (100,101);

Let's check the data in 'product_check' table to see the order in which the trigger is fired.

SELECT * FROM product_check;



Output:

Mesage	Current_Date

Before update, statement level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
After update, statement level	26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement. The above rules apply similarly for INSERT and DELETE statements.

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger. The below statement shows the structure of the view 'USER_TRIGGERS'

DESC USER_TRIGGERS;

NAME	Type

TRIGGER_NAME	VARCHAR2(30)
TRIGGER_TYPE	VARCHAR2(16)
TRIGGER_EVENT	VARCHAR2(75)



TABLE_OWNER	VARCHAR2(30)
BASE_OBJECT_TYPE	VARCHAR2(16)
TABLE_NAME	VARCHAR2(30)
COLUMN_NAME	VARCHAR2(4000)
REFERENCING_NAMES	VARCHAR2(128)
WHEN_CLAUSE	VARCHAR2(4000)
STATUS	VARCHAR2(8)
DESCRIPTION	VARCHAR2(4000)
ACTION_TYPE	VARCHAR2(11)
TRIGGER_BODY	LONG

This view stores information about header and body of the trigger.

```
SELECT * FROM user_triggers WHERE trigger_name =  
'Before_Update_Stat_product';
```

The above sql query provides the header and body of the trigger
'Before_Update_Stat_product'.

You can drop a trigger using the following command.

```
DROP TRIGGER trigger_name;
```