National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Fall 2022, Lab Manual – 09

Course Code: CL-2005	Course : Database Systems Lab
Instructor(s) :	Mafaza

Contents: In this lab, we will discuss Triggers in PL/SQL.

Triggers

A trigger is an event within the DBMS that can cause some code to execute automatically. Triggers are stored programs, which are automatically executed or fired when some events occur. 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).

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

Benefits of Triggers

Triggers can be written for the following purposes:

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

Syntax

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

```
[FOR EACH ROW]
WHEN (condition)
BEGIN
--SQL Statements
END;
```

For Example: 01

To start with, we will be using the EMPLOYEE table:

	ID	NAME	AGE	ADDRESS	SALARY
1	1	Muhammad Nadeem	29	Karachi	2000
2	2	Amin Sadiq	25	Hyedrabad	1500
3	3	Ali Fatmi	23	Lahore	2000
4	4	Fizza Aqeel	25	Islamabad	6500
5	5	Eram Shaheen	27	Peshawar	8500
6	6	Mafaza	22	Faslabad	4500

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 EMPLOYEE table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON employee
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:

OUTPUT

```
Trigger DISPLAY_SALARY_CHANGES compiled
```

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 EMPLOYEE table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO EMPLOYEE (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (7, 'Kariz Kamal', 22, 'Sukkur', 7500.00);
```

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

OUTPUT

```
1 row inserted.
```

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 EMPLOYEE table. The UPDATE statement will update an existing record in the table:

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

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

```
1 row updated.
```

Old salary: 1500

New salary: 2000
Salary difference: 500

For Example: 02

- 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

- **CREATE Table product_price_history**
  (product_id number(5),
  product_name VARCHAR(20),
  supplier_name VARCHAR (20),
  unit_price number(7,2));

- **CREATE TABLE product**
  (product_id number(5),
  product_name VARCHAR(20),
  supplier_name VARCHAR (20),
  unit_price number(7,2))

2) Create the product_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 insert & update the price of a product.

- **Insert into product values (100, 'Laptop', 'Dell', 262.22);**
  Insert into product values (101, 'Laptop', 'HP', 362.22);

- **UPDATE PRODUCT SET unit_price=800 WHERE**
  product_id=100;

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

Types of PL/SQL Triggers

There are two types of triggers based on the 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 Triggers Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level triggers fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

For Example:

- **Let's create a table 'product_check' which we can use to store messages when triggers are fired.**
- **CREATE TABLE product_check**
  (Message VARCHAR(50),
  Current_Date Date
  );

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 update.

- **CREATE or REPLACE TRIGGER Before_Update_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
UPDATE 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:

Message 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

Task

We have simple database that has one table with person information stored in it.

```
CREATE TABLE Person (
ID int NOT NULL,
Name varchar (50) NOT NULL,
PreviousName varchar (50) NULL,
SameNameCount int NULL,
CONSTRAINT pk_Person PRIMARY KEY (ID));
```

The Previous Name column should be updated whenever the value in the Name column is changing with the name before the change. The SameNameCount column should be updated whenever a row is inserted, updated, deleted and should store the number of rows in which the name column has the same value. (E.g. if there are two rows for which the Name value is "Areeba", the SameNameCount column should have the value 2 for both rows.