Practical 9

Aim: Write and execute triggers using PL/SQL.

What is a Trigger?

Triggers in SQL are concise snippets of code that automatically execute when specific events occur on a table. These triggers play a vital role in maintaining data integrity, ensuring the accuracy and consistency of information. Similar to their real-world counterparts, SQL triggers act as a safeguard, responding to events and safeguarding data integrity.

For example, John is the marketing officer in a company. When a new customer data is entered into the company's database he has to send the welcome message to each new customer. If it is one or two customers John can do it manually, but what if the count is more than a thousand? Well in such scenario triggers come in handy.

Thus, now John can easily create a trigger which will automatically send a welcome email to the new customers once their data is entered into the database.

Always remember that there cannot be two triggers with similar action time and event for one table. For example, we cannot have two BEFORE UPDATE triggers for a table. But we can have a *BEFORE UPDATE* and a *BEFORE INSERT* trigger, or a *BEFORE UPDATE* and an *AFTER UPDATE* trigger.

Syntax of Trigger

```
1. Create Trigger Trigger_Name
2. (Before | After) [ Insert | Update | Delete]
3. on [Table_Name]
4. [ for each row | for each column ]
```

5. [trigger_body]

- ***Create Trigger***
  These two keywords are used to specify that a trigger block is going to be declared.
- ***Trigger_Name***
  It specifies the name of the trigger. Trigger name has to be unique and shouldn't repeat.
- **(*Before* | *After*)**
  This specifies when the trigger will be executed. It tells us the time at which the trigger is initiated, i.e, either before the ongoing event or after.
- *Before Triggers* are used to update or validate record values before they're saved to the database.
- *After Triggers* are used to access field values that are set by the system and to effect changes in other records. The records that activate the after trigger are read-only. We cannot use After trigger if we want to update a record because it will lead to read-only error.
- **[*Insert* | *Update* | *Delete*]**
  These are the DML operations and we can use either of them in a given trigger.
- ***on* [*Table_Name*]**
  We need to mention the table name on which the trigger is being applied. Don't forget to use on keyword and also make sure the selected table is present in the database.
- **[for each row | for each column]**
  1. Row-level trigger gets executed before or after *any column value of a row* changes
  2. Column Level Trigger gets executed before or after the *specified column* changes
- **[*trigger_body]***
  It consists of queries that need to be executed when the trigger is called.

We can also create a nested trigger that can do multi-process. Also handling it and terminating it at the right time is very important. If we don't end the trigger properly it may lead to an infinite loop.

Example of the nested trigger: Continuing from the earlier scenario, John sent an email for every new customer that was added to the company's database. Now, what if he

wishes to keep track of the number of customers to whom the email was sent? Now John needs to create a nested trigger to keep the track of the count along with sending an email.

Examples of Triggers in SQL

In the below trigger, we are trying to calculate the percentage of the student as soon as his details are updated to the database.

```
CREATE TRIGGER sample_trigger
before INSERT
ON student
FOR EACH ROW
SET new.total = new.marks/6;
```

```
SQL> CREATE OR REPLACE TRIGGER sample_trigger
  2  BEFORE INSERT ON student
  3  FOR EACH ROW
  4  BEGIN
  5       :NEW.total := :NEW.marks / 6;
  6  END;
  7  /

Trigger created.
```

Here the “NEW” keyword refers to the row that is getting affected.

Operations in Triggers

We can perform many operations using triggers. Some may be simple and some may be a little complex, but once if we go through the query its easy to understand.

● DROP A Trigger

```
DROP TRIGGER trigger name;
```

```
SQL> drop trigger sample_trigger;

Trigger dropped.
```

● Display A Trigger

The below code will display all the triggers that are present.

```
SHOW TRIGGERS;
```

The below code will display all the triggers that are present in a particular database.

```
SHOW TRIGGERS
IN database_name;
```

Example:

```
SHOW TRIGGERS IN sitNagpur;
```

In the above example, all the triggers that are present in the database named

sitNagpur will be displayed.

As we have already understood how to create a trigger, now let's understand the two variants of the trigger those are Before insert and After insert. in order to implement them, let's create a student table with various columns as shown below:

```
CREATE TABLE Student(
studentID INT NOT NULL AUTO_INCREMENT,
FName VARCHAR(20),
LName VARCHAR(20),
Address VARCHAR(30),
City VARCHAR(15),
Marks INT,
PRIMARY KEY(studentID)
);
```

Now if we execute this query we get the following table.

StudentID	Fname	Lname	Address	City	Marks
INT	Varchar(20)	Varchar(20)	Varchar(30)	Varchar(15)	INT

Let's try to use the first variant, i.e, *Before Insert*

```
CREATE TRIGGER calculate
before INSERT
ON student
```

```
FOR EACH ROW
SET new.marks = new.marks+100;
```

```
SQL> CREATE OR REPLACE TRIGGER calculate
  2  BEFORE INSERT ON Student
  3  FOR EACH ROW
  4  BEGIN
  5       :NEW.marks := :NEW.marks + 100;
  6  END;
  7  /

Trigger created.
```

Here when we insert data into the student table automatically, the trigger will be invoked. The trigger will add 100 to the marks column into the student column.

Now let's use the second variant i.e, After Insert

To use this variant we need one more table i.e, Percentage where the trigger will store the results. Use the below code to create the Percentage Table.

```
create table Final_mark(per int );
```

Now let us use the after insert trigger

```
CREATE TRIGGER total_mark
after insert
ON student
```

```
FOR EACH ROW

insert into Final_mark values(new.marks);
```

```
SQL> CREATE OR REPLACE TRIGGER total_mark
  2  AFTER INSERT ON Student
  3  FOR EACH ROW
  4  BEGIN
  5      INSERT INTO Final_mark(mark)
  6      VALUES (:NEW.marks);
  7  END;
  8  /

Trigger created.
```

Here when we insert data to the table, *total_mark trigger* will store the result in the Final_mark table.

Advantages of Triggers

- Forcing security approvals on the table that are present in the database
- Triggers provide another way to check the integrity of data
- Counteracting invalid exchanges
- Triggers handle errors from the database layer
- Normally triggers can be useful for inspecting the data changes in tables
- Triggers give an alternative way to run scheduled tasks. Using triggers, we don't have to wait for the scheduled events to run because the triggers are invoked automatically before or after a change is made to the data in a table

Disadvantages of Triggers

- Triggers can only provide extended validations, i.e., no all kinds of validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN

KEY constraints

- Triggers may increase the overhead of the database
- Triggers can be difficult to troubleshoot because they execute automatically in the database, which may not invisible to the client applications