CSIS05I

Database Systems II

Lab (5)

TRIGGERS

Lab (5)

Triggers:

Triggers are a more complex construct in database systems. They are essentially small functions which are executed when a certain event occurs. When the event in question occurs, the trigger is automatically executed without the user invoking it. In SQL server there are three types of triggers, which are DML, DDL and Logon Triggers. Our scope is the DML Triggers, which are fired automatically in response to DML events; INSERT, UPDATE and DELETE SQL commands.

DML Triggers are broadly divided into 2 groups:

A. AFTER Triggers (also known as FOR Triggers).

B. INSTEAD OF Triggers.

✓ AFTER Triggers:

The After triggers are those that are fired **after** either an INSERT or an UPDATE or a DELETE. In order to illustrate how triggers work, we will create a simple database which would contain at the beginning just one table.

```
CREATE DATABASE TriggerTest;

CREATE TABLE Person(
                    SSN         INT   Primary key,
                    PName       VARCHAR (20),
                    Age         INT
          );
```

We will then insert a few values for the table as follows:

```
INSERT INTO Person
VALUES (1,'Ahmed', 20);

INSERT INTO Person
VALUES (2,'Mohamed', 30);
```

A. AFTER INSERT:

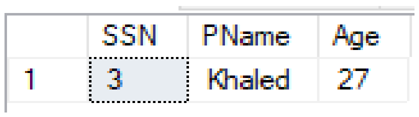
Example 1: Trigger that displays a full record copy of the last inserted record in Person table.

```
Create Trigger PersonInsertingTrigger
ON Person

FOR INSERT
AS

BEGIN

SELECT * FROM inserted;

END
```

Then we will insert a new record to table "Person":

```
INSERT INTO Person
VALUES (3,'Khaled', 27);
```

Output from table "Inserted":



	SSN	PName	Age
1	3	Khaled	27

Please Note that Table "Inserted" is a special custom table to the triggers. It works only within the body of the triggers, in which it keeps only the last inserted record of the table while using the triggers.

Example 2:

In order to illustrate how the AFTER INSERT triggers can help us to keep track with the records insertion logs. We will create extra table that would be treated as a log for the trigger execution called “Auditing” table.

The following code is intended to create a trigger that is to be executed when entering a new record in the Person table. The action that the trigger takes in response to that event occurring is to add a record in the Auditing table that records the new data that has been entered along with a message and a date, as follows:

We will then create the table that would contain the log for the trigger execution:

```
CREATE TABLE Auditing
(
        SSN             INT,
        Audit_msg       VARCHAR (50),
        Audit_date      Date
);
```

When creating triggers, we first need to give them a name and specify the table from which the event would take place. We then specify the type of trigger; in this case it is ‘AFTER INSERT’.

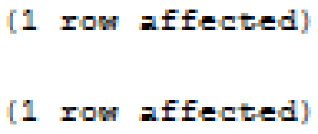
We then declare variables, which would be used to hold the values that are entered in the Person table. The select statements are used to assign values to those variables after which the INSERT statement is executed and the values are re-entered in the Auditing table.

Then we will create our trigger:

```
CREATE TRIGGER PersonAuditTrigger
ON Person
FOR INSERT
AS

BEGIN

DECLARE @ssn INT;
DECLARE @audit_msg VARCHAR (50);

SELECT @ssn = SSN FROM inserted;
SELECT @audit_msg = ' row is inserted ';

INSERT INTO Auditing
VALUES (@ssn, @audit_msg, getdate());

END
```

Then we will now enter a new record in the Person table as follows:

```
INSERT INTO Person
VALUES (4, ' Maged', 18);
```



```
(1 row affected)

(1 row affected)
```

Note that **two rows have been affected** because one record was added to the Person table; consequently a new record was also added to the Auditing table.

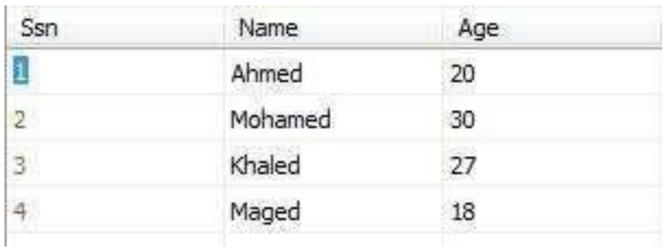
To Test the Trigger, we need to execute the following SELECT statements:

SELECT * FROM Person;

SELECT * FROM Auditing;

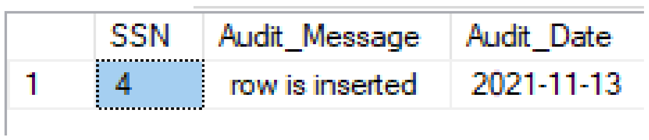
So, the output will be:

Person Table



Ssn	Name	Age
1	Ahmed	20
2	Mohamed	30
3	Khaled	27
4	Maged	18

Auditing Table



	SSN	Audit_Message	Audit_Date
1	4	row is inserted	2021-11-13

B. AFTER DELETE:

The AFTER DELETE Trigger is very close to the After Insert Trigger. As it fires after the Delete statement is executed.

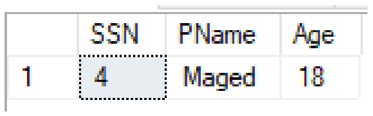
Example 3: Trigger that displays a full record copy of the last deleted record in Person table.

```
Create Trigger PersonDeletingTrigger
ON Person

FOR DELETE
AS

BEGIN

SELECT * FROM deleted;

END
```

Then we will delete a record from table "Person":

```
DELETE FROM Person

WHERE SSN = 4;
```

Output from table "Deleted":



	SSN	PName	Age
1	4	Maged	18

Please Note that Table "Deleted" is a special custom table to the triggers. It works only within the body of the triggers, in which it keeps only the last deleted record of the table while using the triggers.

C. AFTER UPDATE:

The AFTER UPDATE trigger is quite similar to the AFTER INSERT and AFTER DELETE triggers, but instead of writing FOR INSERT, we write FOR UPDATE. The command that is executed after executing the trigger is modified accordingly and thus becomes an UPDATE statement.

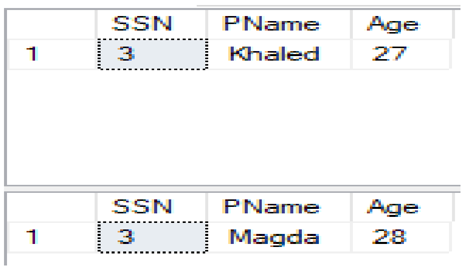
Example 4: Trigger that displays full record copies of the old and the new version of an updated record in Person table.

```
Create Trigger PersonUpdatingTrigger
ON Person

FOR UPDATE
AS

BEGIN

SELECT * FROM deleted;
SELECT * FROM inserted;

END
```

Then we will update a record in table "Person":

```
UPDATE Person SET
PName = 'Magda', Age = 28
WHERE SSN = 3;
```

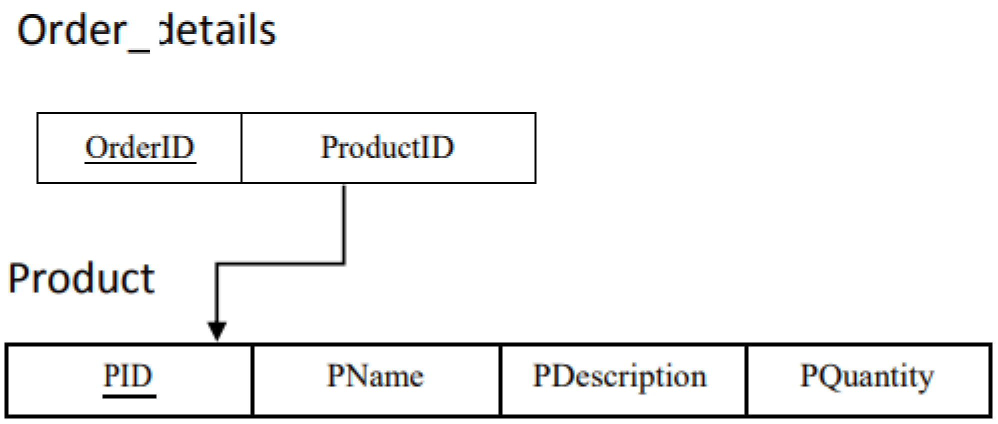
Output from table "Deleted and Inserted":



```
   SSN  PName   Age
1  3    Khaled  27

   SSN  PName   Age
1  3    Magda   28
```

Note: Trigger also used to change in the records of table. In which you can fire a trigger on a certain table and at the same time it affects the records values of another table.

Example 5 (Store Database): For example if I asked you, whenever you will get an order you need to change in the quantity of this specific product. So now, if we want to deduct the quantity by one in table product anytime you get an order. Therefore, we need to do a trigger on table order_details and update table product with the new quantity.



```
Create Table Product
( PID               INT            Primary key,
  PName             Varchar(20),
  PDescription      Varchar(30),
  PQuantity         INT
);

Create Table Order_Details
( OrderID           INT             Primary key,
  ProductID         INT,

    constraint Order_Details_fk foreign key (ProductID)
    references Product (PID)
);

Insert into Product
Values(1, 'Camera', 'Web Camera' , 30);
```

```
Insert into Order_Details
Values(101,1);



CREATE TRIGGER ProductOrderUpdateTrigger
ON order_details

FOR INSERT
AS

UPDATE Product
SET PQuantity= PQuantity-1
FROM inserted, product
WHERE inserted.productID= product.PID;
```

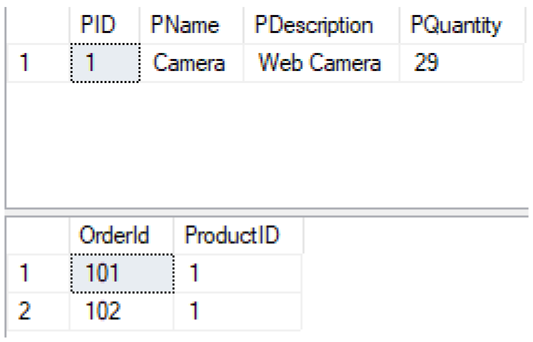
Then we will insert a new record to table order_details that will affect table Product directly:

```
Insert into order_details
Values(102, 1);
```

To Test the Trigger:

```
Select * From Product;
Select * From Order_Details;
```

Output:



	PID	PName	PDescription	PQuantity
1	1	Camera	Web Camera	29

	OrderId	ProductID
1	101	1
2	102	1

Example 6: Create trigger that eliminates adding more than or equals to five products for each order, in the order_details table. The user will receive an error message and their attempt to add more than or equal to five products per order (for the same order) will be denied.

Note: The tables attributes used in this question are different from our normal schema.

```
CREATE TABLE Product(
PID INT,
PName VARCHAR(30),

CONSTRAINT Product_pk PRIMARY KEY (PID)
);

CREATE TABLE Orderr(
OrderID INT,
OrderName varchar(30),

CONSTRAINT Order_pk PRIMARY KEY (OrderID),

);


CREATE TABLE Order_Details(
ProductID       INT,
OrderID         INT,
Quantity        INT,

CONSTRAINT Orderdetails_pk PRIMARY KEY (OrderID, ProductID),
CONSTRAINT Order_Product_fk FOREIGN KEY (OrderID) REFERENCES Orderr
(OrderID),
CONSTRAINT Product_Order_fk FOREIGN KEY (ProductID) REFERENCES Product
(PID)
);
```

```
Create or Alter trigger T1
on Order_Details
For Insert
As

if (      select count (*)
          from Order_Details O Join inserted I
          On O. OrderID  = I. OrderID
   ) >=5

Begin

Raiserror('This order cannot be done, as you have exceeded your product quantity limit
per order!!!', 16, 1)

rollback;

End


--5 Products--

insert into Product values(1,'P1');

insert into Product values(2,'P2');

insert into Product values(3,'P3');

insert into Product values(4,'P4');

insert into Product values(5,'P5');

--Only one Order--

insert into Orderr values(10,'Order10');

--Inserting Products to the Same Order--

insert into Order_Details values(1, 10, 20);

insert into Order_Details values(2, 10, 30);

insert into Order_Details values(3, 10, 2);

insert into Order_Details values(4, 10, 50);
```

```
--Testing the fifth line with the Same Order--

insert into Order_Details values(5, 10, 20);

--The Raise Error message will appear here then the insertion will not be executed as we
used 'Rollback'--

--Testing the Insertions after executing the trigger--

select * from Order_Details;
```

- Triggers can be disabled for specific table as follows:

ALTER TABLE Person **DISABLE TRIGGER** ALL;

The triggers can be enabled again by replacing the keyword **DISABLE** with **ENABLE**. You may also specify a specific trigger by providing its name rather than writing **ALL.**

✓ INSTEAD OF Triggers (Self-study for extra knowledge):

The **INSTEAD OF** Triggers fire instead of the triggering action/event itself in all DML cases such as insert or update or delete. So, the Instead of Trigger fires before SQL Server starts the execution of the action that fired it. This is different from the AFTER trigger that fires after the action that caused it to fire. We can have an INSTEAD OF insert/update/delete trigger on a table that successfully executed.

In the previous lab, we mentioned the views and how to retrieve and insert records using it. The view is a virtual table based on the result-set of an SQL statement but if a view is based on Multiple Tables, and if you update or insert through this view, it may not update the underlying base tables correctly. So to correctly update/ insert a view that is based on multiple tables, INSTEAD OF Triggers should be used.

So, **INSTEAD OF** Triggers are very important to overcome this famous Views Limitation. In which we can't insert directly into a view that is connecting two different table that have a common relationship, as this will lead to compilation errors while the SQL engine try to update multiple tables as the same time. Thus, it will results in raising an error by the SQL server (`View is not updatable because the modification affects multiple base tables`). This Views Limitation can only be solved by using the INSTEAD OF Triggers.

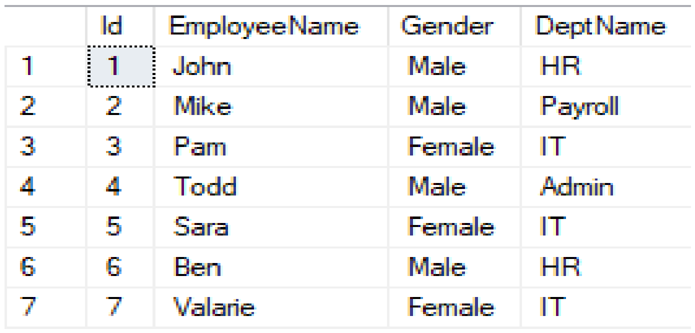
Example 7:

Write a **Trigger** that enables the user to insert a view called “**vWEmployeeDetails**” that has attributes (Id, EmployeeName, Gender and DeptName). In which this view joins two tables Table **Employee** (Id, EmployeeName, Gender and DepartmentId) and Table **Department** (DeptId, DeptName). In addition, make sure that the user will not be able to insert non-existing departments or NULL values, otherwise raise an error message to the user.

```
CREATE TABLE Employee
(
Id int Primary Key,
EmployeeName varchar(30),
Gender  varchar(10),
DepartmentId int
);

CREATE TABLE Department
(
DeptId int Primary Key,
DeptName varchar(20)
);

Insert into Department values (1,'IT');
Insert into Department values (2,'Payroll');
Insert into Department values (3,'HR');
Insert into Department values (4,'Admin');

Insert into Employee values (1,'John', 'Male', 3);
Insert into Employee values (2,'Mike', 'Male', 2);
Insert into Employee values (3,'Pam', 'Female', 1);
Insert into Employee values (4,'Todd', 'Male', 4);
Insert into Employee values (5,'Sara', 'Female', 1);
Insert into Employee values (6,'Ben', 'Male', 3);

--creating the view
go
Create view vWEmployeeDetails
as
Select Id, EmployeeName, Gender, DeptName
from Employee join Department
on Employee.DepartmentId = Department.DeptId
go
```

```
--testing the view
Select * from vWEmployeeDetails;

--error occurs!--
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT');

--creating the InsteadOf Insert Trigger

Create trigger tr_vWEmployeeDetails_InsteadOfInsert
on vWEmployeeDetails
Instead Of Insert
as

Begin
Declare @DeptId int

--Check if there is a valid Department Id for the given Department Name the user entered

Select @DeptId = DeptId
from Department join inserted
on inserted.DeptName = Department.DeptName

--If Department Id is null throw an error and stop processing (this means that the user
entered wrong or garbage department name)

if(@DeptId is null)
Begin
Raiserror('Invalid Department Name. Statement terminated', 16, 1)
return
End

--Finally insert into tblEmployee table using the help of table inserted!

Insert into Employee(Id, EmployeeName, Gender, DepartmentId)
Select Id, EmployeeName, Gender, @DeptId
from inserted
End

--Testing the Trigger!
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT');

--Testing the View again!
Select * from vWEmployeeDetails;
```

Output:



	Id	EmployeeName	Gender	DeptName
1	1	John	Male	HR
2	2	Mike	Male	Payroll
3	3	Pam	Female	IT
4	4	Todd	Male	Admin
5	5	Sara	Female	IT
6	6	Ben	Male	HR
7	7	Valarie	Female	IT

Exercises

1. Create Triggers that keep track with both the inserted and the deleted records of table "Person" to be saved and displayed from a log table called "Auditing Table".

2. Create a Trigger that enables the user to insert in a view called "StMajorView" that has attributes (StudentID, StudentName, and MajorName). In which this view joins two tables; Table Student (StudentID, StudentName and MajorNo) and Table Major (MajorID and MajorName). In addition, make sure that the user will not be able to insert non-existing majors or NULL values, otherwise raise an error message to the user **(Self-study for extra knowledge).**