ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Database Systems

Chapter 5
T-SQL Programming

Session 2:
Function, Store procedure, Trigger

http://vku.udn.vn/

Outline

Functions

- A function is a set of SQL statements that perform a specific task.
- The main purpose of function is code reusability.
  - If you have to repeatedly write large SQL scripts to perform the same task, you can create a function that performs that task.
  - Next time instead of rewriting the SQL, you can simply call that function.
- **A function accepts inputs in the form of parameters and returns a value**.

Functions

- Some rules when creating functions in SQL Server
  - A function must have an unique name
  - Functions only work with select statements
  - Functions compile every time.
  - Functions must return a value or result.
  - Functions only work with input parameters.
  - Try and catch statements are not used in functions

Functions

- SQL Server supports two types of functions:
  - User-Defined function (also call UDF): 3 types
    - **Scalar Valued Functions:** returns a single value
    - **Inline table-valued function:** returns a table
    - **Multi-statement table-valued function**: returns a table and can have more than one T-SQL statement.
  - System Defined function (Built-in function)
- Syntax

```
CREATE FUNCTION [schema_name.]function_name(parameter_list)
RETURNS data_type
AS
      BEGIN statements
            RETURN value
      END
```

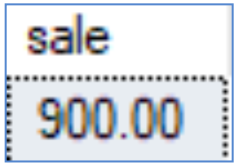
Scalar Valued Functions

❑Example:

```
CREATE FUNCTION fn_Sale(
    @quantity INT, @list_price DEC(10,2), @discount DEC(4,2))
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * (1 - @discount)
END
```

❑Calling a scalar function

```
SELECT
dbo.fn_Sale(10,100,0.1) sale
```

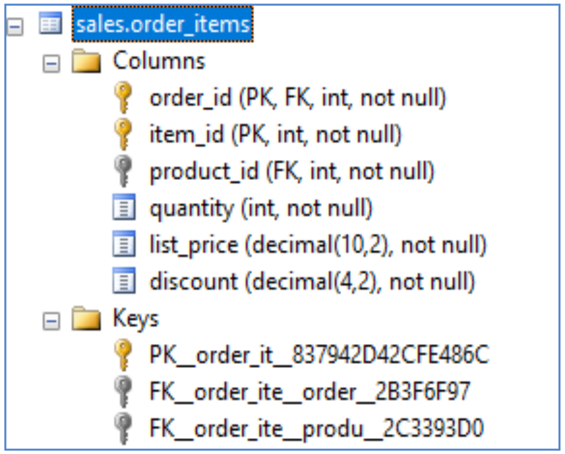


```
sale
900.00
```

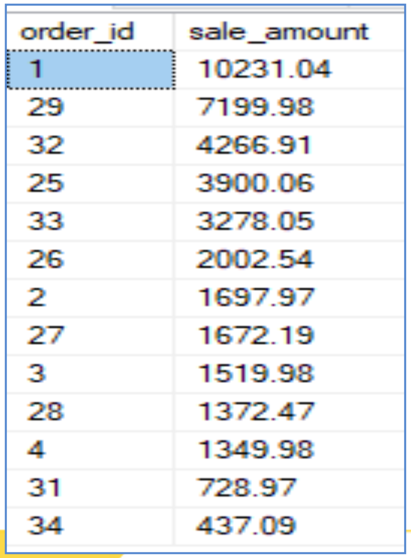
Scalar Valued Functions

❑Calling the function to get the sale of sales of sales orders in the order_items table

```
SELECT
    order_id,
    SUM(dbo.fn_Sale(quantity,
list_price, discount)) sale_amount
FROM
    sales.order_items
GROUP BY
    order_id
ORDER BY
    sale_amount DESC
```



```
sales.order_items
  Columns
    order_id (PK, FK, int, not null)
    item_id (PK, int, not null)
    product_id (FK, int, not null)
    quantity (int, not null)
    list_price (decimal(10,2), not null)
    discount (decimal(4,2), not null)
  Keys
    PK__order_it__837942D42CFE486C
    FK__order_ite__order__2B3F6F97
    FK__order_ite__produ__2C3393D0
```



order_id	sale_amount
1	10231.04
29	7199.98
32	4266.91
25	3900.06
33	3278.05
26	2002.54
2	1697.97
27	1672.19
3	1519.98
28	1372.47
4	1349.98
31	728.97
34	437.09

Inline Table-Valued Functions

- Returns data of a table type whose values is derived from a single SELECT statement.
  - Example

```
CREATE FUNCTION fn_getProducts()
RETURNS TABLE
AS
RETURN SELECT * FROM products
```

  - Executing an inline table-valued functions

```
SELECT * FROM fn_getProducts()
```

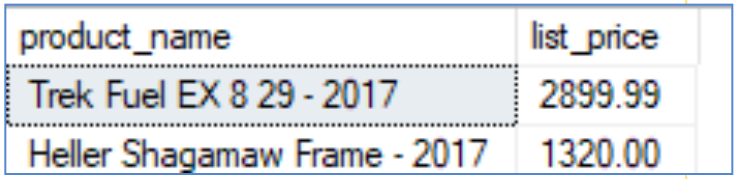
Inline Table-Valued Functions

- Example the function requires input parameters

```
CREATE FUNCTION fn_ProductInYear(@model_year INT)
RETURNS TABLE
AS
RETURN
    SELECT product_name,model_year, list_price
    FROM products
    WHERE model_year = @model_year
```

- Executing an inline table-valued functions

```
SELECT
    product_name, list_price
FROM fn_ProductInYear(2017)
```



product_name	list_price
Trek Fuel EX 8 29 - 2017	2899.99
Heller Shagamaw Frame - 2017	1320.00

Multi-statement table-valued function

- The function can take one or more paramenters and returns a table.
- You must define the table structure that is being returned.
- After creating this type of user-defined function, we can **use it in the FROM clause of a T-SQL command.**

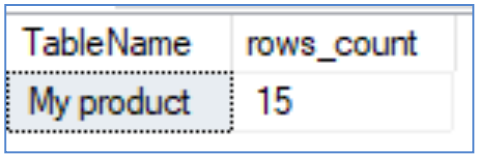
Multi-statement table-valued function

❑Example

```
CREATE FUNCTION fn_rowOfTables() RETURNS
@table table (TableName varchar(50), rows_count int)
AS
BEGIN
       DECLARE  @num int
       SELECT @num = count(product_id) FROM products
       INSERT INTO @table values('My product', @num)
       IF @@ROWCOUNT =0
       BEGIN
          INSERT INTO @table VALUES('', 'No row is added')
       END
       RETURN
END
```

❑Executing a table-valued functions

```
SELECT * FROM fn_rowOfTables()
```



TableName	rows_count
My product	15

Alter/drop a function

- You can modify an existing scalar function by ALTER FUNCTION

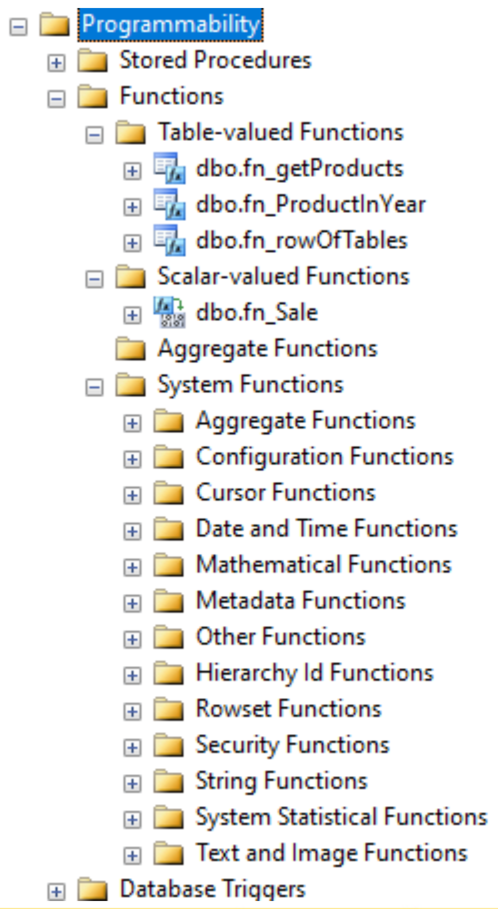
```
ALTER FUNCTION [schema_name.]function_name(parameter_list)
RETURNS data_type
AS
      BEGIN statements
            RETURN value
      END
```

- You can delete an existing scalar function by DROP FUNCTION

```
DROP FUNCTION [schema_name.]function_name
```

Functions

- You can you can find the functions under Programmability -> Functions of the database that you're working



```
⊟ Programmability
  ⊞ Stored Procedures
  ⊟ Functions
    ⊟ Table-valued Functions
      ⊞ dbo.fn_getProducts
      ⊞ dbo.fn_ProductInYear
      ⊞ dbo.fn_rowOfTables
    ⊟ Scalar-valued Functions
      ⊞ dbo.fn_Sale
      Aggregate Functions
    ⊟ System Functions
      ⊞ Aggregate Functions
      ⊞ Configuration Functions
      ⊞ Cursor Functions
      ⊞ Date and Time Functions
      ⊞ Mathematical Functions
      ⊞ Metadata Functions
      ⊞ Other Functions
      ⊞ Hierarchy Id Functions
      ⊞ Rowset Functions
      ⊞ Security Functions
      ⊞ String Functions
      ⊞ System Statistical Functions
      ⊞ Text and Image Functions
  ⊞ Database Triggers
```

System Defined function

❑System Defined function or Built-in function. Some common functions:

- **CAST** – cast a value of one type to another.
- **CONVERT** – convert a value of one type to another.
- CHOOSE – return one of the two values based on the result of the first argument.
- ISNULL – replace NULL with a specified value.
- ISNUMERIC – check if an expression is a valid numeric type.
- **IIF** – add if-else logic to a query.

System Defined function

- TRY_CAST – cast a value of one type to another and return NULL if the cast fails.
- TRY_CONVERT – convert a value of one type to another and return the value to be translated into the specified type. It returns NULL if the cast fails.
- TRY_PARSE – convert a string to a date/time or a number and return NULL if the conversion fails.
- **Convert datetime to string** – show you how to convert a datetime value to a string in a specified format.
- **Convert string to datetime** – describe how to convert a string to a datetime value.
- **Convert datetime to date** – convert a datetime to a date.

Example

- Convert datetime to string
  - Syntax

```
CONVERT(VARCHAR, datetime [,style])
```

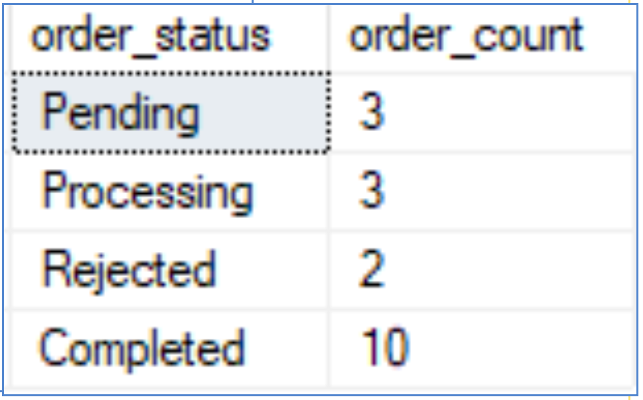
  - Example: display the current date with dd/mm/yyyy format

```
SELECT CONVERT(VARCHAR, GETDATE(),103) AS 'Now'
```

Example

- Using IFF function
  - Example: returns the corresponding order status based on the status number in order table

```sql
SELECT
    IIF(order_status = 1,'Pending',
        IIF(order_status=2, 'Processing',
            IIF(order_status=3, 'Rejected',
                IIF(order_status=4,'Completed','N/A')
            )
        )
    ) order_status,
    COUNT(order_id) order_count
FROM orders
GROUP BY
    order_status;
```



order_status	order_count
Pending	3
Processing	3
Rejected	2
Completed	10

Store Procedures

❑SQL Server stored procedure (SP) is a batch of statements grouped as a logical unit and stored in the database.

❑Procedure is stored in cache area of memory when the stored procedure is created so that **it can be used repeatedly**. SQL Server **does not have to recompile it every time the stored procedure is run**.

❑It can accept input parameters, return output values as parameters, or return success or failure status messages

Store Procedures vs SQL

SQL Statement

First Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Second Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Stored Procedure

Creating
- *Check syntax*
- *Compile*

First Time
- *Execute*
- *Return data*

Second Time
- *Execute*
- *Return data*

Store Procedures

- Benefits of SP
  - Reusable
  - It can be easily modified
  - Performance
  - Reduced network traffic
  - Security

Store Procedures

- ❑There are two types of stored procedures available in SQL Server:
    - User defined stored procedures
    - System stored procedures

Store Procedures

- Naming conventions for stored procedures
  - It is a good idea to come up with a standard prefix to use for your stored procedures: usp_ , sp , usp…
  - Do not use sp_ as a prefix
    - This is a standard naming convention that is used in the master database
  - Give the action that the stored procedure takes and then give it a name representing the object it will affect.
    - uspInsertProduct
    - uspGetProductById
    - spValidateProduct
  - Consider using the schema that you will use when saving the objects. The schema is useful if you want to keep all utility like objects together

Store Procedures

- Example: create a store procedure that returns a list of products from the products table

```
CREATE PROCEDURE uspProductList
AS
BEGIN
    SELECT product_name, list_price
    FROM products
    ORDER BY product_name
END2
```

- Executing the sp `EXECUTE sp_name` or `EXEC sp_name`

```
EXEC uspProductList
```

Store Procedures

- Modifying an existing stored procedure
  - By using the ALTER PROCEDURE statement.

```
ALTER PROCEDURE uspProductList
AS
BEGIN
    SELECT *
    FROM products
    ORDER BY product_name
END
```

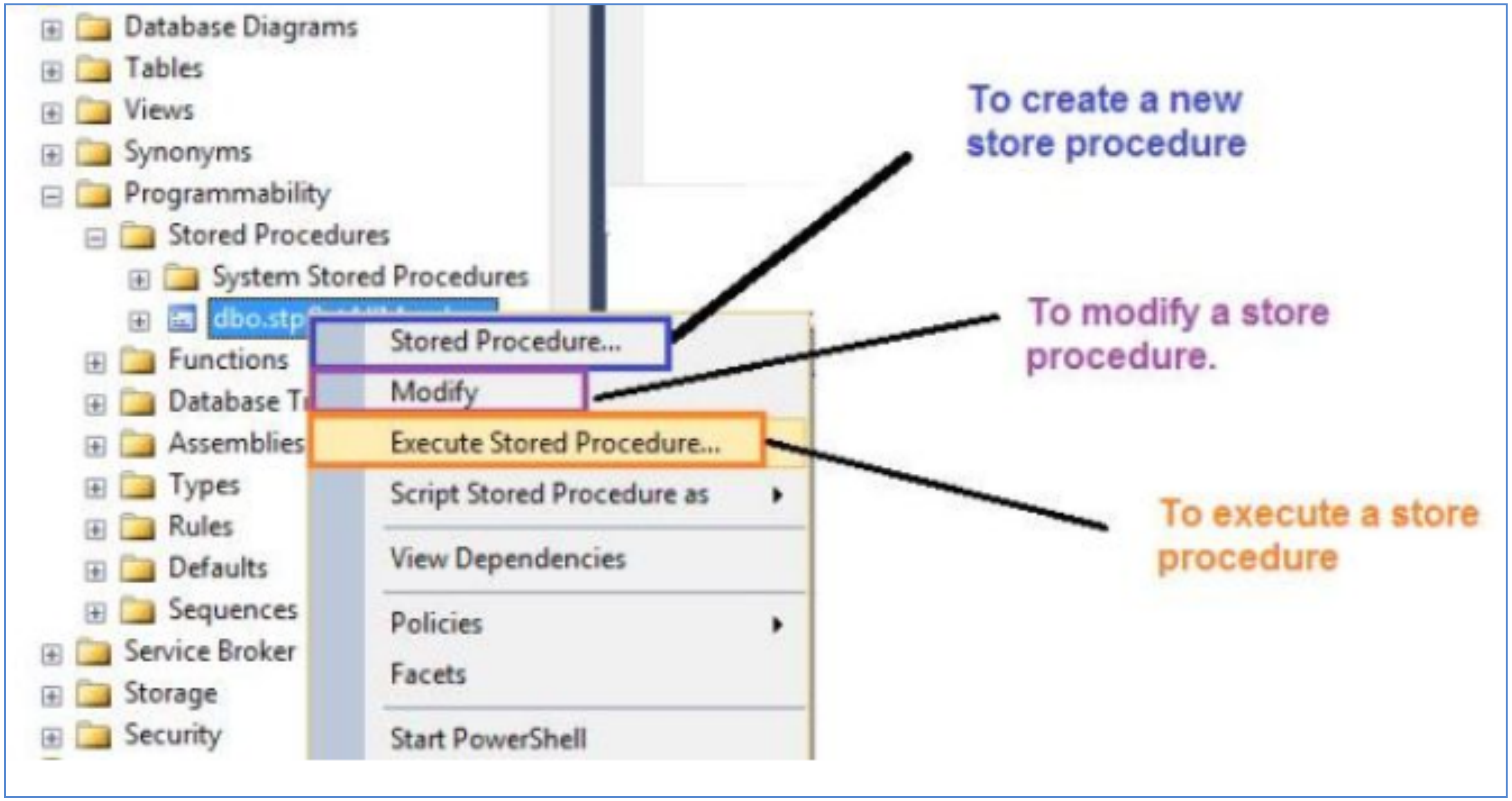
- Deleting a stored procedure
  - using the DROP PROCEDURE or DROP PROC statement:

```
DROP PROCEDURE sp_name
```

```
DROP PROCEDURE sp_name
```

Store Procedures

❑From the UI for SP



Store Procedures

- ❑ Parameters in SPs are used to pass input values and return output values. There are two types of parameters:
  - Input parameters – By default, pass values to a stored procedure.
  - Output parameters - Return values from a stored procedure, use OUTPUT keyword

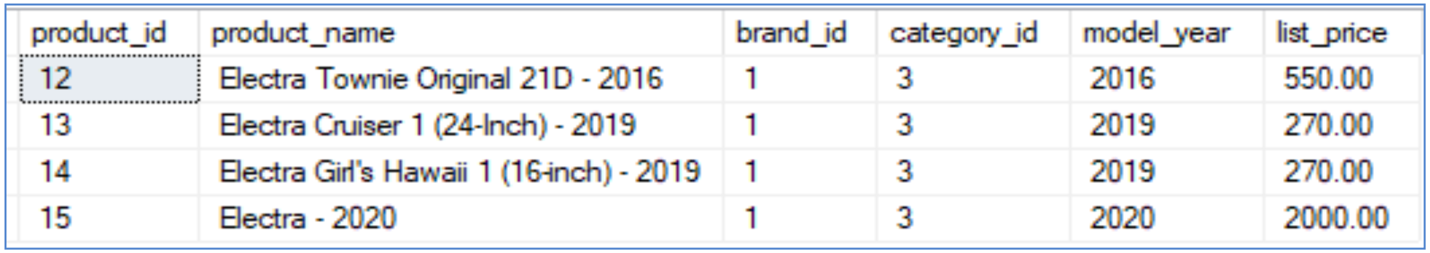
Store Procedures

- Example: SELECT query SP with parameters

```
CREATE PROCEDURE uspGetProductByName
    @productName nvarchar(30)
AS
BEGIN
    SET NOCOUNT ON
    SELECT * FROM products
    WHERE product_name LIKE '%'+@productName+'%'
END
```

- Executing the SP

```
EXEC uspGetProductByName @productName = 'Electra'
```



product_id	product_name	brand_id	category_id	model_year	list_price
12	Electra Townie Original 21D - 2016	1	3	2016	550.00
13	Electra Cruiser 1 (24-Inch) - 2019	1	3	2019	270.00
14	Electra Girl's Hawaii 1 (16-inch) - 2019	1	3	2019	270.00
15	Electra - 2020	1	3	2020	2000.00

Store Procedures

❑ Example: INSERT query SP with parameters

```
CREATE PROC uspInsertProduct
@category_id INT, @brand_id INT,
@pro_name VARCHAR(50), @year INT,
@pro_price DECIMAL(10,2) = 0
AS
BEGIN
       DECLARE @checkExist int
       SELECT @checkExist = category_id FROM products
       WHERE category_id = @category_id
       IF  (@checkExist IS NULL)
         BEGIN
              PRINT 'This product category does not exist in system!'
         RETURN
       END
INSERT INTO products
VALUES (@pro_name,@brand_id,@category_id, @year, @pro_price)
END
```

Store Procedures

- Executing the SP

```
EXEC uspInsertProduct @category_id =6, @brand_id = 1,
                                 @pro_name = 'Heller 2020',@year = 2020,
                                 @pro_price = 1000
```

```
(1 row(s) affected)
```

- Default Parameter Values
  - In most cases it is always a good practice to pass in all parameter values, but sometimes it is not possible.
  - So you can assign value to parameter with NULL or an valid value
  - So with default parameter, you don’t need to pass in a parameter value when you execute the SP

Store Procedures

- Exercises: create and execute store procedures:
  - To update/delete a products
  - To find products with product price whose list prices are in range of min and max list prices and the product name also contain a piece of text that you pass in.
  - To insert an order and order items tables

Store Procedures

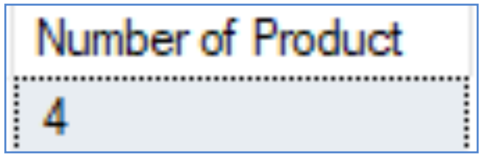
❑SP with OUTPUT parameter

```sql
CREATE PROCEDURE uspGetProductCount
@productName nvarchar(30), @productCount int OUTPUT
AS
SELECT @productCount = count(*)
FROM products
WHERE product_name LIKE @productName +'%'
```

Store Procedures

- Executing the SP with OUTPUT parameter
  - First we are going to declare a variable, execute the stored procedure
  - Then select the returned valued.

```sql
DECLARE @product_Count INT
EXEC uspGetProductCount @productName= 'Su',
@productCount = @product_Count OUTPUT
SELECT @product_Count AS 'Number of Product'
```



Number of Product
4

Store Procedures

- You can use TRY-CATCH statement with error handling in SQL Server
- Example

```
CREATE PROCEDURE uspTryCatchTest
AS
BEGIN TRY
      SELECT 1/0
END TRY
BEGIN CATCH
      SELECT ERROR_NUMBER() AS ErrorNumber
      ,ERROR_SEVERITY() AS ErrorSeverity
      ,ERROR_STATE() AS ErrorState
      ,ERROR_PROCEDURE() AS ErrorProcedure
      ,ERROR_LINE() AS ErrorLine
      ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

Trigger

- A trigger is a special type of stored procedure that is **executed automatically** as part of a data modification.
- A trigger is created on a table and associated with one or more actions linked with a data modification (INSERT, UPDATE, or DELETE).
- When one of the actions for which the trigger is defined occurs, the trigger fires automatically

Trigger

- Following are some examples of trigger uses:
  - Maintenance of duplicate and derived data
  - Complex business rules
  - Cascading referential integrity
  - Complex defaults
  - Implement complex security authorizations
- Disadvantages of trigger
  - It increases the overhead of the database server.
  - Providing an extended validation, not replacing all the validation which can be done only by the application layer.
  - SQL triggers are executed from the client applications, which will be challenging to figure out what is happening in the database layer.

Trigger

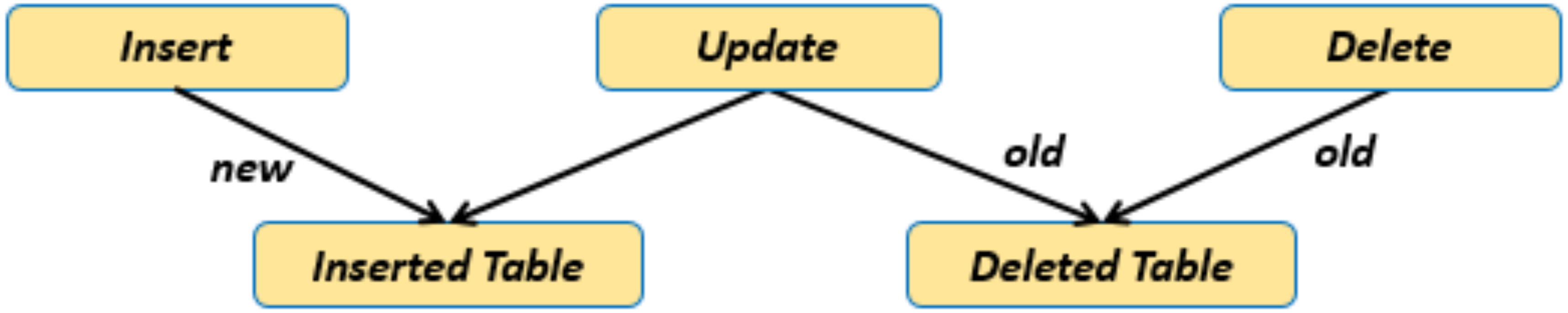
- There are two types of triggers:
  - DDL (Data Definition Language) triggers: triggers fires upon events that change the structure (like creating, modifying or dropping a table)
  - DML (Data Modification Language) triggers. This is the most used class of triggers. The firing event is a data modification statement; it could be an insert, update or delete statement either on a table or a view. DML triggers have different types:
    - **FOR or AFTER [INSERT, UPDATE, DELETE]**: These types of triggers are executed after the firing statement ends (either an insert, update or delete).
    - **INSTEAD OF [INSERT, UPDATE, DELETE]**: the INSTEAD OF triggers executes instead of the firing statement.

Deleted and Inserted tables

- ❑When you create a trigger, you have access to two temporary tables (the **deleted and inserted tables**).
- ❑They are referred to as tables, but they are different from true database tables. They are stored in memory—not on disk.

Deleted and Inserted tables

- When the insert, update or delete statement is executed. All data will be copied into these tables with the same structure.

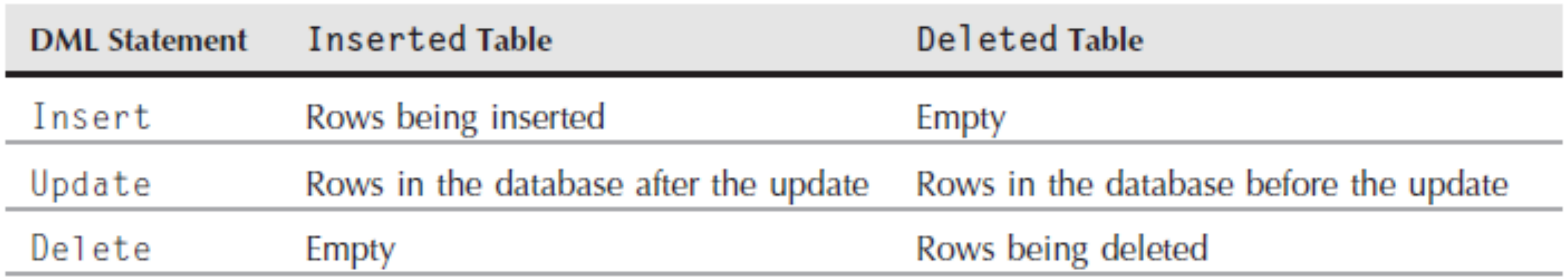


- The values in the inserted and deleted tables are accessible only within the trigger. Once the trigger is completed, these tables are no longer accessible.

Deleted and Inserted tables

- The following table shows the content of the INSERTED and DELETED tables before and after each event:

Inserted and Deleted Tables



DML Statement	Inserted Table	Deleted Table
Insert	Rows being inserted	Empty
Update	Rows in the database after the update	Rows in the database before the update
Delete	Empty	Rows being deleted

Trigger

- Syntax

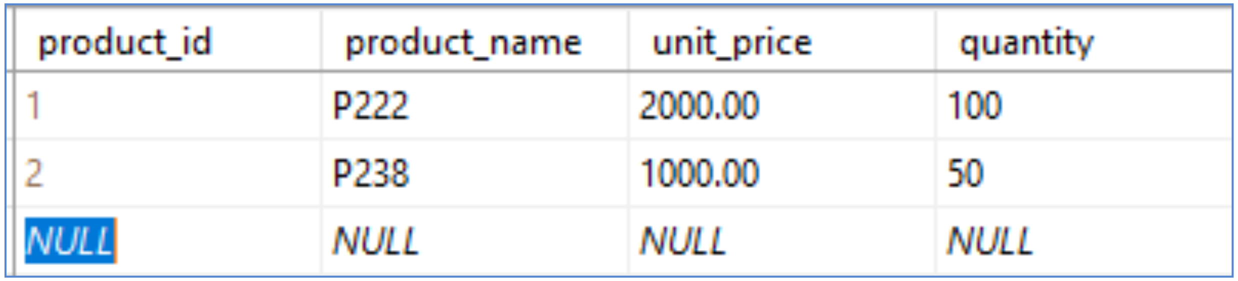
```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
{FOR | AFTER | INSTEAD OF} {[INSERT] [,] [UPDATE] [,]
[DELETE]}
AS
{sql_statements}
```

- [before | after]: This specifies when the trigger will be executed.
- {insert | update | delete}: This specifies the DML operation.
- on [table_name]: This specifies the name of the table associated with the trigger.
- [sql_statements]: is one or more Transact-SQL used to carry out actions once an event occurs.

DML Trigger Example

- Product table

```
CREATE TABLE Product
(
   product_id INT PRIMARY KEY IDENTITY,
   product_name varchar(30) NOT NULL,
   unit_price DECIMAL(10,2) NOT NULL,
   quantity INT NOT NULL
)
```



product_id	product_name	unit_price	quantity
1	P222	2000.00	100
2	P238	1000.00	50
NULL	*NULL*	*NULL*	*NULL*

DML Trigger Example

- Product_history table

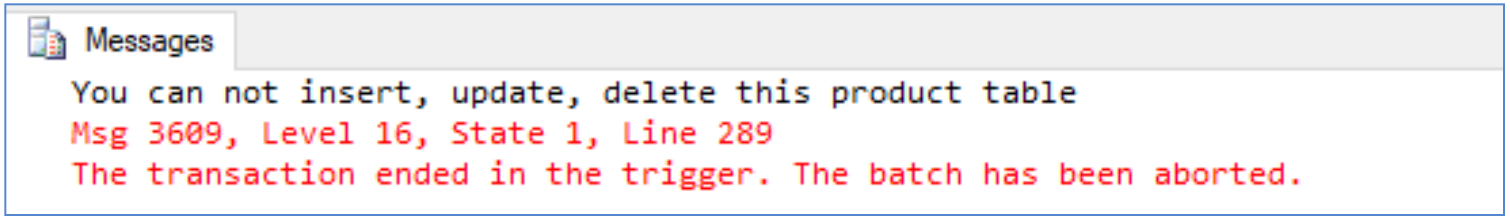
```
CREATE TABLE Product_history
(
   Id INT PRIMARY KEY IDENTITY,
   Product_history varchar(200) NOT NULL,
   update_at DATETIME NOT NULL,
   operation VARCHAR(30) NOT NULL
      CHECK (operation IN ('INSERT','DELETE'))
)
```

DML Trigger Example

- Create a simple trigger that prevent DML on product table

```
CREATE TRIGGER triggerTestDML
ON product
FOR INSERT, UPDATE, DELETE
AS
   PRINT 'You can not insert, update, delete this product table'
   ROLLBACK
GO
```

- When we insert, update or delete in a table in a database then the following message appears



Messages

```
You can not insert, update, delete this product table
Msg 3609, Level 16, State 1, Line 289
The transaction ended in the trigger. The batch has been aborted.
```

DML Trigger Example

❑INSERT trigger

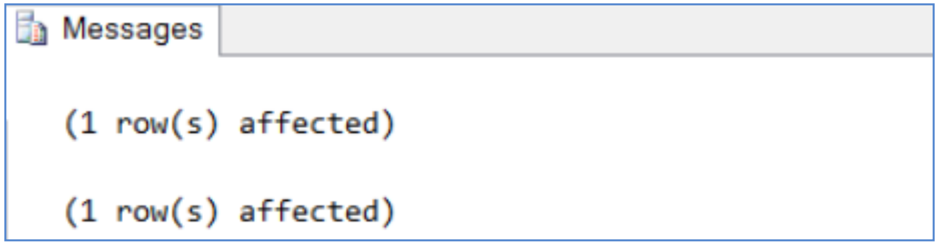
```
CREATE TRIGGER UTRG_Product_Insert
ON  Product
FOR INSERT
AS
BEGIN
       DECLARE @product_id int, @product_name varchar(40),
@unit_price DECIMAL(10,2)
       SELECT @product_id = product_id, @product_name =
product_name, @unit_price = unit_price FROM inserted
       INSERT INTO Product_history VALUES('New product with Id '
+ cast (@product_id As varchar(40)),GETDATE(), 'INSERT')
END
```

```
INSERT INTO Product VALUES('P555',
1000,20)
```

DML Trigger Example

```
INSERT INTO Product VALUES('P555',1000,20)
```

- After insert a row into Product table, the trigger will be occur.

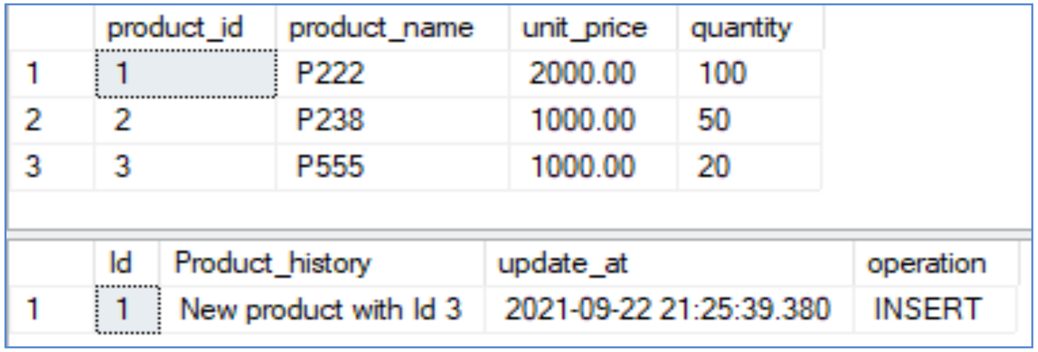


Messages

```
(1 row(s) affected)

(1 row(s) affected)
```

```
SELECT * FROM dbo.Product
SELECT * FROM dbo.Product_history
```



	product_id	product_name	unit_price	quantity
1	1	P222	2000.00	100
2	2	P238	1000.00	50
3	3	P555	1000.00	20

	Id	Product_history	update_at	operation
1	1	New product with Id 3	2021-09-22 21:25:39.380	INSERT

DML Trigger Example

- Exercises
  - Create trigger for Update/Delete?
  - Or create 1 trigger for INSERT/UPDATE/DELETE event occur against the product table in one trigger?

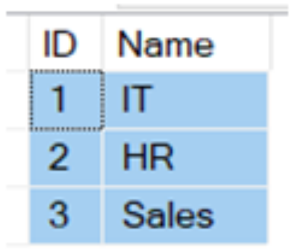
INSTEAD OF trigger

- ❑The INSTEAD OF triggers are the DML triggers that are fired instead of the triggering event such as the INSERT, UPDATE or DELETE events.
- ❑So, when you fire any DML statements such as Insert, Update, and Delete, then on behalf of the DML statement, the instead of trigger is going to execute.
- ❑In real-time applications, Instead Of Triggers are used to correctly update a complex view.

INSTEAD OF trigger

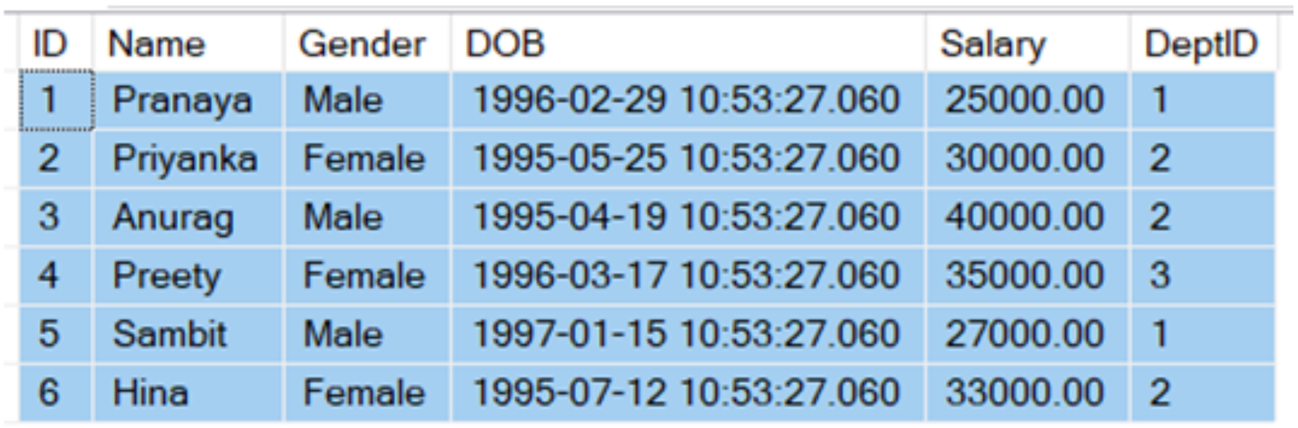
- Example: Department and Employee tables

Department



ID	Name
1	IT
2	HR
3	Sales

Employee



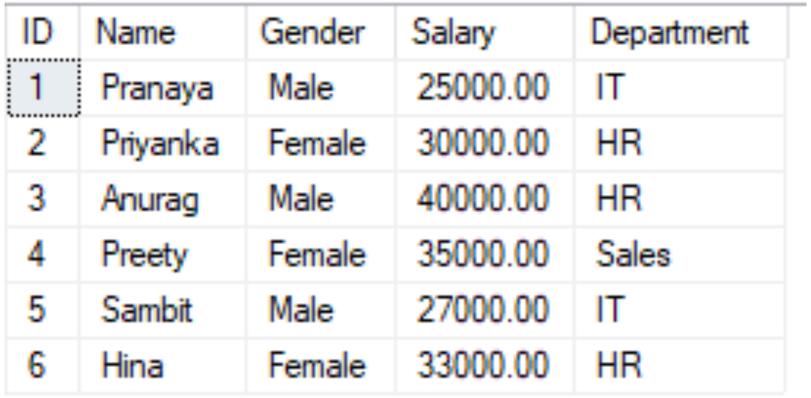
ID	Name	Gender	DOB	Salary	DeptID
1	Pranaya	Male	1996-02-29 10:53:27.060	25000.00	1
2	Priyanka	Female	1995-05-25 10:53:27.060	30000.00	2
3	Anurag	Male	1995-04-19 10:53:27.060	40000.00	2
4	Preety	Female	1996-03-17 10:53:27.060	35000.00	3
5	Sambit	Male	1997-01-15 10:53:27.060	27000.00	1
6	Hina	Female	1995-07-12 10:53:27.060	33000.00	2

INSTEAD OF trigger

- Let create a view

```
CREATE VIEW vwEmployeeDetails
AS
SELECT emp.ID, emp.Name, Gender, Salary, dept.Name AS Department
FROM Employee emp
INNER JOIN Department dept
ON emp.DeptID = dept.ID
```

- Get data from the view



ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	IT
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR

INSTEAD OF trigger

- Insert a record into the view vwEmployeeDetails by executing the following query.

```
INSERT INTO vwEmployeeDetails VALUES(7, 'Saroj', 'Male',
50000, 'IT')
```

- Will recieve an error message: 'View or function vwEmployeeDetails is not updatable because the modification affects multiple base tables.'

=>Use INSTEAD OF trigger

INSTEAD OF trigger

```sql
CREATE TRIGGER tr_vwEmployeeDetails_InsteadOfInsert
ON vwEmployeeDetails
INSTEAD OF INSERT
AS
BEGIN
  DECLARE @DepartmenttId int
   -- First Check if there is a valid DepartmentId in the Department Table for
the given Department Name
  SELECT @DepartmenttId = dept.ID
  FROM Department dept
  INNER JOIN INSERTED inst
  on inst.Department = dept.Name
  --If the DepartmentId is null then throw an error
  IF(@DepartmenttId is null)
  BEGIN
    RAISERROR('Invalid Department Name. Statement terminated',16, 1)
    RETURN
  END
   -- Finally insert the data into the Employee table
  INSERT INTO Employee(ID, Name, Gender, Salary, DeptID)
  SELECT ID, Name, Gender, Salary, @DepartmenttId
  FROM INSERTED
End
```

After executing the trigger, the record is inserted into the view and the Employee table.

INSTEAD OF trigger

- Exercises
  - Create INSTEAD OF trigger for Update/Delete operations

Thank You !