Fpt University

Chapter 8: Database programming on SQL Server

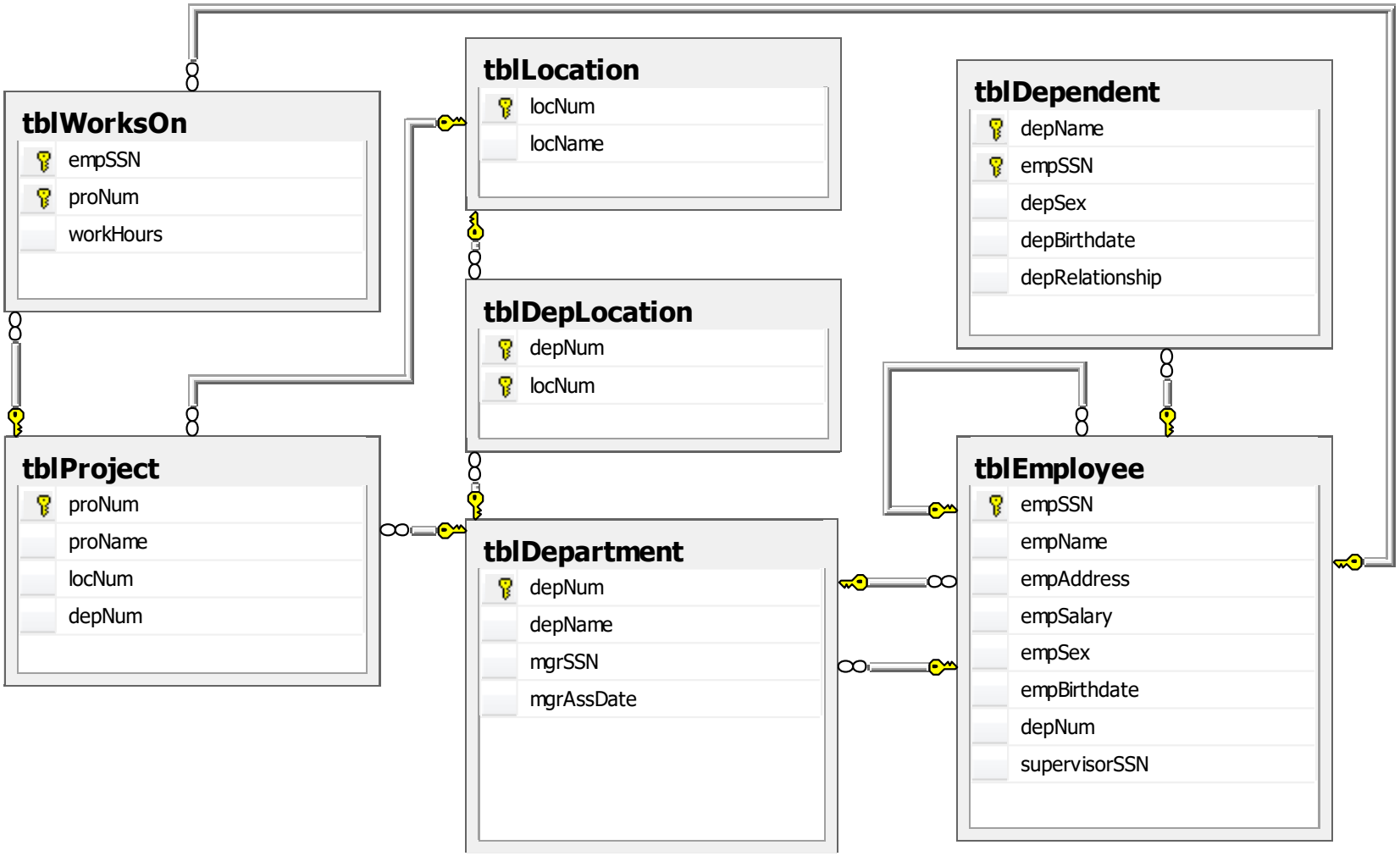
Objectives

- Understand what triggers are for and how to use
- Understand what stored-procedure are for and how to use
- Understand what cursors are for and how to use
- Understand what functions are for and how to use
- Understand the difference between T-SQL programming with other programming languages
- Understand the useful of trigger, function, stored-procedure (compared with SQL statements)

Contents

- T-SQL Programming
- Stored-procedure
- Functions
- Triggers
- Cursors

Physical Diagram - FUHCompany



Short introduction to T-SQL programming

1. Variables

- Declare a variable

```
DECLARE  @local_variable [AS]  data_type [=initialvalue] , …
```

data_type: any system-supplied, common language runtime (CLR) user-defined table type. A variable cannot be of text, ntext, or image data type

- Example

```
DECLARE    @empName NVARCHAR(20), @empSSN AS DECIMAL,
        @empSalary DECIMAL=1000
```

Short introduction to T-SQL programming

1. Variables (cont)

- Assign a value into a variable : using SET or SELECT

```
SET @empName=N'Mai Duy An'
SELECT @empSalary=2000
```

- Assign a value into a variable using SQL command : SELECT or UPDATE

```
SELECT  @empName=empName, @empSalary=empSalary
FROM    tblEmployee
WHERE   empName=N'Mai Duy An'

UPDATE  tblEmployee
SET @empName=empName, @empSalary=empSalary
WHERE   empName=N'Mai Duy An'
```

Short introduction to T-SQL programming

1. Variables (cont)

- Display value of a variable : using PRINT or SELECT

```
PRINT   @empName
SELECT  @empSalary
```

- Converts an expression from one data type to a different data type : using CAST or CONVERT function

```
DECLARE @empName NVARCHAR(20), @empSalary DECIMAL
SET @empName=N'Mai Duy An'
SET @empSalary=1000
PRINT @empName + '''s salary is ' + CAST(@empSalary AS VARCHAR)
PRINT @empName + '''s salary is ' + CONVERT(VARCHAR, @empSalary)
```

Short introduction to T-SQL programming

2. Flow-control statement

- Statement Blocks: Begin…End
- Conditional Execution:
  - ✓IF ... ELSE Statement
  - ✓CASE ... WHEN
- Looping: WHILE Statement
- Error handling:
  - ✓@@ERROR
  - ✓TRY … CATCH

Short introduction to T-SQL programming

Statement Blocks: BEGIN…END

- Groups of statements used with IF, WHILE, and CASE statements must be grouped together using the BEGIN and END statements. Any BEGIN must have a corresponding END in the same batch.

IF ... ELSE Statement

- evaluate a Boolean expression and branch execution based on the result

```
DECLARE @workHours DECIMAL, @bonus DECIMAL
SELECT  @workHours=SUM(workHours)
FROM    tblWorksOn
WHERE   empSSN=30121050027
GROUP BY empSSN

IF (@workHours > 300)
    SET @bonus=1000
ELSE
    SET @bonus=500
PRINT @bonus
```

Short introduction to T-SQL programming

CASE ... WHEN Statement

- Syntax

```
CASE input_expression
    WHEN  when_expression  THEN result_expression
    [WHEN when_expression  THEN result_expression…n]
    [ELSE else_result_expression ]
END
```

- Example

```
DECLARE @depNum DECIMAL, @str NVARCHAR(30)
SET @str=
    CASE @depNum
        WHEN 1 THEN N'Phòng ban số 1'
        WHEN 2 THEN N'Phòng ban số 2'
        ELSE N'Mã phòng ban khác 1, 2'
    END
PRINT @str
```

Short introduction to T-SQL programming

We use CASE in statements such as SELECT, UPDATE, DELETE and SET, and in clauses such as SELECT list, IN, WHERE, ORDER BY, and HAVING

```
DECLARE @womanDayBonus DECIMAL

SELECT @womanDayBonus =
    CASE empSex
        WHEN 'F' THEN 500
        WHEN `M' THEN 0
    END
FROM tblEmployee
WHERE empSSN=30121050004

PRINT @womanDayBonus
```

Short introduction to T-SQL programming

WHILE Statement : repeats a statement or block of statements as long as a specified condition remains true

- Syntax

```
WHILE boolean_expression
    SQL_statement | block_of_statements
    [BREAK]
    SQL_statement | block_of_statements
    [CONTINUE]
```

- Example

```
DECLARE @factorial INT, @n INT
SET @n=5
SET @factorial=1
WHILE (@n > 1)
    BEGIN
        SET @factorial = @factorial*@n
        SET @n = @n - 1
    END
PRINT @factorial
```

Short introduction to T-SQL programming

Handling error using @@ERROR function

- The @@ERROR system function returns 0 if the last Transact-SQL statement executed successfully; if the statement generated an error, @@ERROR returns the error number

```
BEGIN TRANSACTION
    INSERT INTO tblDepartment(depNum,depName)
        VALUES(6, N'Phòng Kế Toán');

    INSERT INTO tblDepartment(depNum,depName)
        VALUES(6, N'Phòng Kế Toán');

    IF @@ERROR<>0
        BEGIN
            ROLLBACK TRANSACTION
            PRINT @@ERROR
        END
COMMIT TRANSACTION
```

Short introduction to T-SQL programming

Handling error using TRY … CATCH

- was introduced with SQL Server 2005. Statements to be tested for an error are enclosed in a BEGIN TRY…END TRY block. A CATCH block immediately follows the TRY block, and error-handling logic is stored here

```
BEGIN TRANSACTION--begin transaction
BEGIN TRY
     --oparations
     INSERT INTO tblDepartment(depNum,depName)
          VALUES(6, N'Phòng Kế Toán');

     INSERT INTO tblDepartment(depNum,depName)
          VALUES(6, N'Phòng Kế Toán');

     COMMIT TRANSACTION    --commit the transaction
END TRY
BEGIN CATCH
     ROLLBACK TRANSACTION  --rollback transaction
     PRINT ERROR_NUMBER()
     PRINT ERROR_MESSAGE()
END CATCH
```

Branching Statements

If statement

- Ends with keyword END IF
- If-statement nested within the else-clause are introduced with the single word ELSEIF

```
IF <condition> THEN
        <statement list>
ELSEIF <condition> THEN
        <statement list>
ELSEIF
        …
ELSE
        <statement list>
END IF;
```

Queries in T-SQL programming

Several ways that select-from-where queries are used in PSM

- Subqueries can be used in conditions, or in general, any place a subquery is legal in SQL
- Queries that return a single value can be used as the right sides of assignment statements
- A single-row select statement is a legal statement in PSM
- We can declare and use a cursor for embedded SQL

Loops in T-SQL programming

The basic loop construct in PSM is

```
LOOP
    <statement list>
END LOOP;
```

It is possible to break out of the loop

```
LEAVE <loop label>;
```

Example

```
loop1: LOOP
    . . .
    LEAVE loop1;
    . . .
END LOOP;
```

Other Loop Constructs

```
WHILE <condition> DO
    <statement list>
END WHILE;
```

```
REPEAT <statement list>
UNTIL <condition>
END REPEAT;
```

Exceptions in T-SQL programming

The form of a handler declaration is

```
DECLARE <where to go next> HANDLER FOR
<condition list> <statement list>;
```

The choices for *where to go next*

- CONTINUE
- EXIT
- UNDO

The three – tier Architecture

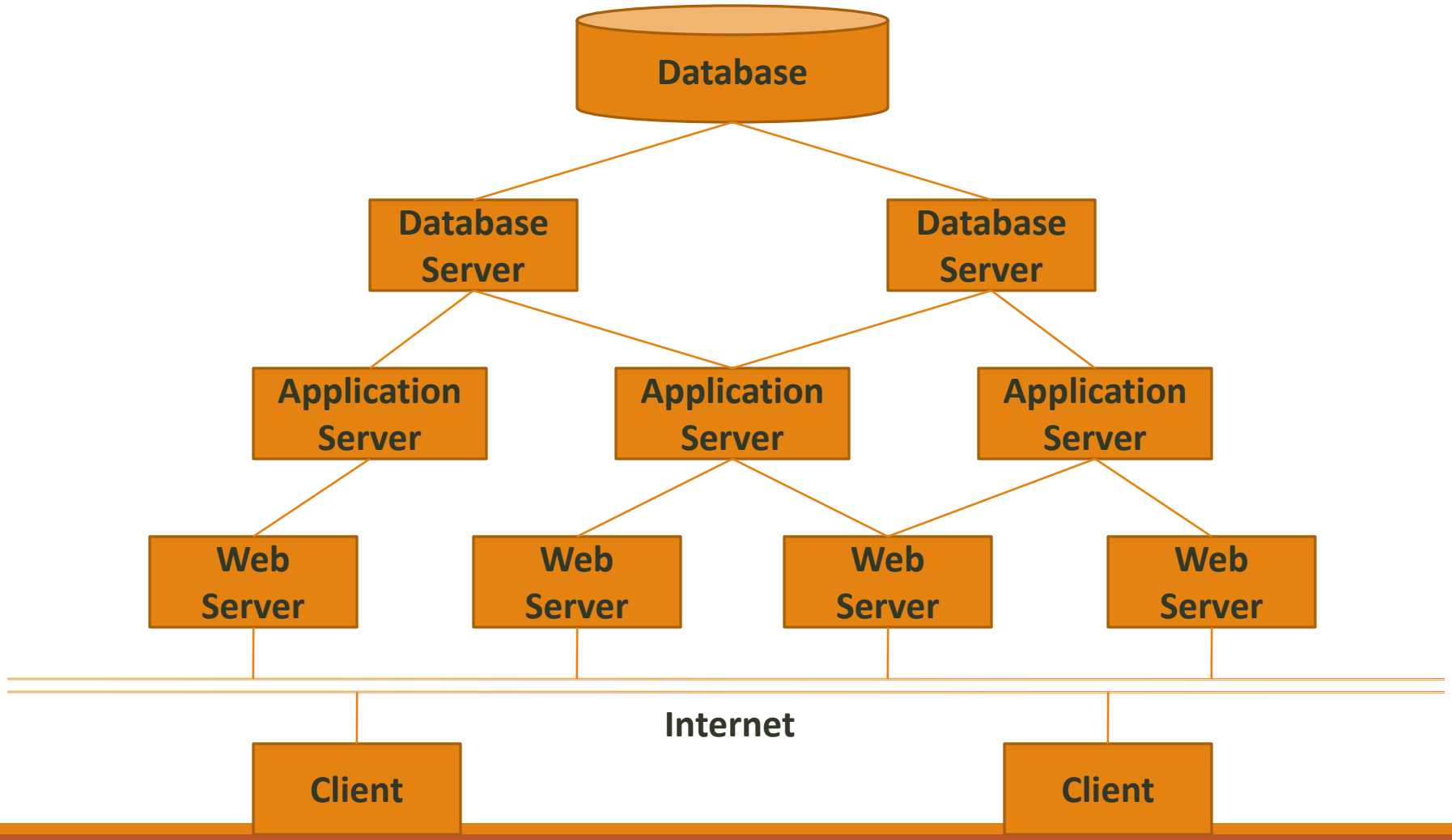
A very common architecture for large database installation

Three different, interacting functions

- Web servers
- Application servers
- Database servers

The processes can run on the same processor or on a large number of processors

The three – tier Architecture



The Webserver Tier

The webserver processes manage the interactions with the user

When a user makes contact, a webserver response to the request, and the user becomes a *client* of this webserver process

The Application Tier

Turning data from the database into a response to the request that it receives from the webserver

One webserver process invoke many application-tier processes, which can be on one or many different machines

The application-tier processes execute the business logic of the organization operating the database

The Database Tier

There can be many processes in the database tier

The processes can be in one or many machines

The database tier executes queries that are requested from the application tier

Advantages of using Stored Procedure

Using stored procedures offer numerous advantages over using SQL statements. These are:

- Reuse of Code
- Maintainability
- Reduced Client/Server Traffic
- Precompiled Execution
- Improved Security

Stored procedure - Introduction

Persistent, Stored Modules (SQL/PSM)

Help to write procedures in a simple, general-purpose language and to store them in the database

We can use these procedures in SQL queries and other statements to perform computations

Each commercial DBMS offers its own extension of PSM

Creating Stored Procedure under MS SQL Server

Create stored procedure:

```
CREATE PROCEDURE procedure_name
    [ {@parameter1 data_type} [= default] [OUTPUT] ]
    [ {@parameter2 data_type} [= default] [OUTPUT] ]
    ...
AS
    sql_statement1
  sql_statement2
```

Calling stored procedure

```
EXEC procedure_name [argument1, argument2,
  …]
```

Creating PSM Functions and Procedures

Example 1:

- Create stored procedure to list all projects
- Create stored procedure to change the project's name
- Create stored function to return the name of project

Example

```
/*
/////////LISTING ALL PROJECTS/////////////////////////////.
*/
IF OBJECT_ID ( 'psm_List_ALL_Of_Project', 'P' ) IS NOT NULL
    DROP PROCEDURE psm_List_ALL_Of_Project;
GO
CREATE PROCEDURE psm_List_ALL_Of_Project
AS
    SELECT *
    FROM tblProject;
GO
EXEC psm_List_ALL_Of_Project;
```

Example

```
/*
////////////////////////////////////////////////////////////////.
*/
IF OBJECT_ID ( 'psm_Change_Name_Of_Project', 'P' ) IS NOT NULL
    DROP PROCEDURE psm_Change_Name_Of_Project;
GO
CREATE PROCEDURE psm_Change_Name_Of_Project
    @PNUMBER INT,
    @PNAME NVARCHAR(50)
AS
    UPDATE tblProject
    SET proNAME=@PNAME
    WHERE proNum=@PNUMBER;
GO

EXEC psm_Change_Name_Of_Project 1,'ProjectA';
GO
```

Function in SQL Server

- System Defined Function
- User Defined Function
  - Scalar functions
  - Inline table-valued functions
  - Multi-statement table-valued functions

Scalar functions

```
CREATE FUNCTION FunctionName
(
        @parameter1 Datatype,
        @Parameter2 datatype,
        @parametern datatype
)
Returns <Return attribute Data type>
AS
BEGIN
        --Function Body
         Return <Return data type>
End
```

Calling a Function in SQL Server

```
SELECT dbo.<FunctionName>(Value)
```

//demo

Inline Table-valued Function

```
-- Syntax for creating an Inline table value function
CREATE FUNCTION Function_Name
(
    @Param1 DataType,
    @Param2 DataType,
    @ParamN DataType
)
RETURNS TABLE
AS
RETURN (Select_Statement)

-- Syntax for calling an Inline table value function
SELECT * FROM Function_Name (VALUE)
```

//demo

Multi-Statement Table Valued Function

```
-- Syntax For Creating Multi Statement Table valued Function
CREATE FUNCTION FunctionName
(
        @Param1 DataType,
        @Param2 DataType,
        @Paramn DataType
)
RETURNS @TableVariable TABLE (Column_Definitions)
WITH FunctionAttribute
AS
BEGIN
        FunctionBody
        RETURN
END
```

//demo

Triggers

Triggers differ from the other constraints

- Triggers are only awakened when certain events occur (INSERT, UPDATE, DELETE)
- One awakened, the trigger tests a condition.
  - If the condition does not hold, trigger do nothing to response to occurred event
  - If the condition is satisfied, the action associated with trigger is performed by the DBMS

Why uses triggers

Triggers can implement business rules

- E.g. creating a new Order when customer checkout a shopping cart (in online ecommerce websites)

Triggers be used to ensure data integrity

- E.g. Updating derived attributes when underlying data is changed, or maintaining summary data

Triggers in SQL

Some principle features of triggers

- The check of trigger's condition and the action of the trigger may be executed either on the state of database that exists before the triggering event is itself executed or on the state that exists after the triggering event is executed
- The condition and action can refer to both old and/or new values of tuples that were updated in the triggering event
- It is possible to define update events that are limited to a particular attribute or set of attributes
- Trigger executes either
  - Once for each modified tuple
  - Once for all the tuples that are changed in one SQL statement

Triggers in SQL (standard)

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
  OLD ROW AS OldTuple,
  NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
  UPDATE MovieExec
  SET netWorth=OldTuple.netWorth
  WHERE cert#=NewTuple.cert#;
```

The Options for Trigger Design

AFTER/BEFORE

UPDATE/INSERT/DELETE

WHEN (<condition>)

OLD ROW/NEW ROW

BEGIN … END;

FOR EACH ROW/FOR EACH STATEMENT

Implement Trigger with T-SQL

Implement Trigger with T-SQL

Create Trigger on MS SQL Server syntax

```
CREATE TRIGGER trigger_name ON TableName
    {AFTER {[DELETE] [,] [INSERT] [,] [UPDATE]}
 AS
    sql_statement 1
    sql_statement 2
        …
```

Disable a trigger

```
DISABLE  TRIGGER <trigger_name> ON <table_name>
```

Enable a trigger

```
ENABLE TRIGGER <trigger_name> ON <table_name>
```

Implement Trigger with T-SQL Samples

Create the trigger raised after insert on tblEmployee table

```
IF OBJECT_ID('Tr_Employee_Insert', 'TR') is not null
    drop trigger Tr_Employee_Insert
go
CREATE TRIGGER Tr_Employee_Insert ON tblEmployee
AFTER INSERT
AS
    RAISERROR('Insert trigger is awakened',16,1)
go
```

- Using AFTER INSERT, UPDATE to raise the trigger after INSERT or UPDATE action

Implement Trigger with T-SQL

Transaction Management in Triggers

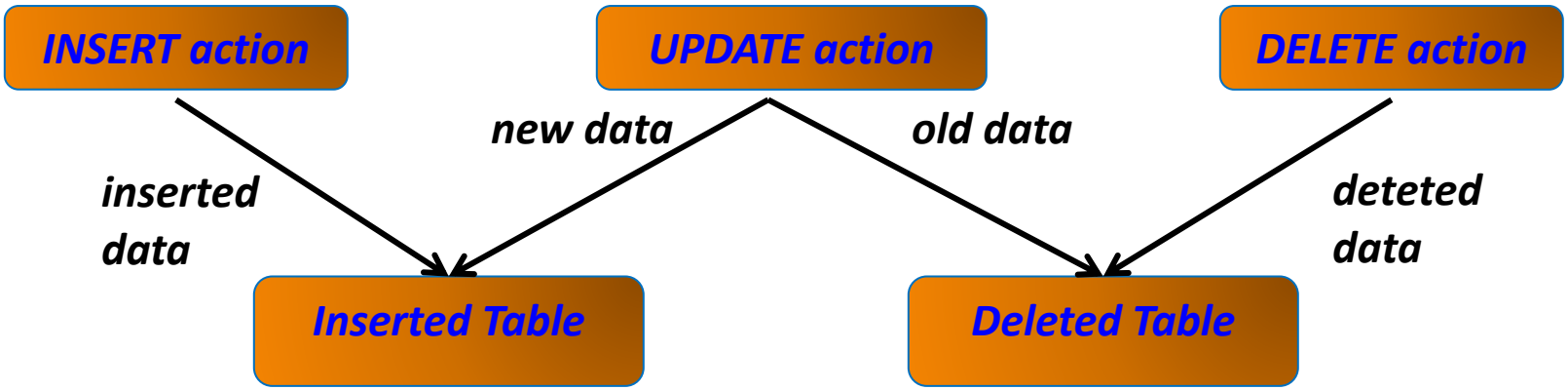
- A trigger is always part of the transaction that initiates it. That transaction can be explicit (when SQL Server has executed Begin Transaction). It can also be implicit basically (SQL Server treats each Transact-SQL statement as a separate transaction)

```
CREATE TRIGGER Tr_Employee_Insert ON tblEmployee
AFTER INSERT
AS
    RAISERROR('Insert trigger is awakened',16,1)
    ROLLBACK TRANSACTION
go
--test
INSERT INTO tblEmployee(empSSN, empName, empSalary, depNum)
VALUES (30121050345, N'Nguyễnn Văn Tý', 10000, 1;
--not found employee whose empSSN is 30121050345
SELECT * FROM tblEmployee WHERE empSSN=30121050345
```

Implement Trigger with T-SQL

Deleted and Inserted tables

- When a trigger is executing, it has access to two memory-resident tables that allow access to the data that was modified: **Inserted** and **Deleted**.
- These tables are available only within the body of a trigger for read-only access.
- The structures of the inserted and deleted tables are the same as the structure of the table on which the trigger is defined



Implement Trigger with T-SQL

Example: using Deleted and Inserted tables

```
IF OBJECT_ID('Tr_Employee_Insert', 'TR') is not null
    drop trigger Tr_Employee_Insert
go
CREATE TRIGGER Tr_Employee_Insert ON tblEmployee
AFTER INSERT
AS
    DECLARE @vEmpSSN DECIMAL, @vEmpName NVARCHAR(50)
    SELECT @vEmpSSN=empSSN FROM inserted
    SELECT @vEmpName=empName FROM inserted
    PRINT 'new tuple:'
    PRINT 'empSSN=' + CAST(@vEmpSSN AS nvarchar(11)) + '
empName=' + @vEmpName
go

--test
INSERT INTO tblEmployee(empSSN, empName, empSalary, depNum,
supervisorSSN)
VALUES (30121050345, N'Nguyễn Văn Tý', 10000, 1,
30121050037);
```

Samples

Create the trigger that refuses all under-18-year-old employee's insertion or update

```
CREATE TRIGGER Tr_Employee_Under18 ON tblEmployee
AFTER INSERT, UPDATE
AS
    DECLARE @empBirthdate DATETIME, @age INT
    SELECT @empBirthdate=empBirthdate
    FROM inserted

    SET @age=YEAR(GETDATE()) - YEAR(@empBirthdate)
    IF (@age < 18)
    BEGIN
        RAISERROR('Employee is under 18 years old.
                We can not sign a contact with
                him/her.',16,1)
        ROLLBACK TRANSACTION
    END
go
```

Samples

Another method: using EXISTS

```
CREATE TRIGGER Tr_Employee_Under18 ON tblEmployee
AFTER INSERT, UPDATE
AS
    IF EXISTS(SELECT *
        FROM inserted
        WHERE (YEAR(GETDATE())-YEAR(empBirthdate))<18
    )
    BEGIN
        RAISERROR('Employee is under 18 years old.
                We can not sign a contact with
                him/her.',16,1)
        ROLLBACK TRANSACTION
    END
go
```

Using Cursor in MS SQL Server

1. Declare cursor

   DECLARE cursor_name CURSOR FOR SELECT Statement
2. Open cursor

   OPEN cursor_name
3. Loop and get values of each tuple in cursor with FETCH statement

   FETCH NEXT | PRIOR | FIRST | LAST
   FROM cursor_name INTO @var1, @var2
4. Using @@FETCH_STATUS to check fetch status. The 0 value mean FETCH statement was successful.
5. CLOSE cursor_name
6. DEALLOCATE cursor_name

Example

```
DECLARE @SSN DECIMAL, @FULLNAME NVARCHAR(50), @message NVARCHAR(200)
DECLARE employee_cursor CURSOR
FOR SELECT empSSN, empName FROM tblEmployee
OPEN employee_cursor
FETCH NEXT FROM employee_cursor INTO @SSN,@FULLNAME
IF @@FETCH_STATUS <> 0
    PRINT '          <<None>>'
WHILE @@FETCH_STATUS = 0
BEGIN
    SELECT @message = '          '+@FULLNAME
    PRINT @message
    FETCH NEXT FROM employee_cursor INTO @SSN,@FULLNAME
END
CLOSE employee_cursor
DEALLOCATE employee_cursor
```

Example

```
IF OBJECT_ID ( 'psm_Change_Of_Project', 'P' ) IS NOT NULL
    DROP PROCEDURE psm_Change_Of_Project;
GO
CREATE PROCEDURE psm_Change_Of_Project
    @dep1 INT,
    @dep2 INT,
    @loc2 NVARCHAR(50),
    @dep3 INT,
    @loc3 NVARCHAR(50)
AS
    DECLARE @pnum INT, @locname NVARCHAR(50)
    DECLARE pro_cursor CURSOR FOR SELECT p.proNum,l.locName
                                  FROM tblProject p, tblLocation l
                                  WHERE p.locNum=l.locNum AND p.depNum = @dep1;

    OPEN pro_cursor;
    FETCH NEXT FROM pro_cursor INTO @pnum,@locname
    IF @@FETCH_STATUS <> 0
        PRINT '         <<None>>'
    WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @locname = @loc2
            UPDATE tblProject SET depNum=@dep2 WHERE proNum=@pnum;
        ELSE IF @locname = @loc3
            UPDATE tblProject SET depNum=@dep3 WHERE proNum=@pnum;

        FETCH NEXT FROM pro_cursor INTO @pnum,@locname
    END
CLOSE pro_cursor
DEALLOCATE pro_cursor
GO
EXEC psm_Change_Of_Project 2,1,N'TP Hà Nội',3,N'TP Hồ Chí Minh';
GO
```

Example

```
DECLARE @EmployeeID DECIMAL(18,0);
DECLARE @EmployeeName NVARCHAR(50), @SALARY DECIMAL(10,0);
DECLARE myCursor CURSOR FOR
                 SELECT empSSN, empName, empSalary
                 FROM tblEMPLOYEE
                 WHERE depNum=1;
OPEN myCursor;
FETCH NEXT FROM myCursor INTO @EmployeeID,@EmployeeName,@Salary;
IF @@FETCH_STATUS <> 0
    PRINT '                <<NONE>>';
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT cast(@EmployeeID as nvarchar(50))+'              '+
    @EmployeeName+'            '+cast(@Salary as nvarchar(50));
    FETCH NEXT FROM myCursor INTO @EmployeeID,@EmployeeName,@Salary;
END
CLOSE myCursor;
DEALLOCATE myCursor;
```