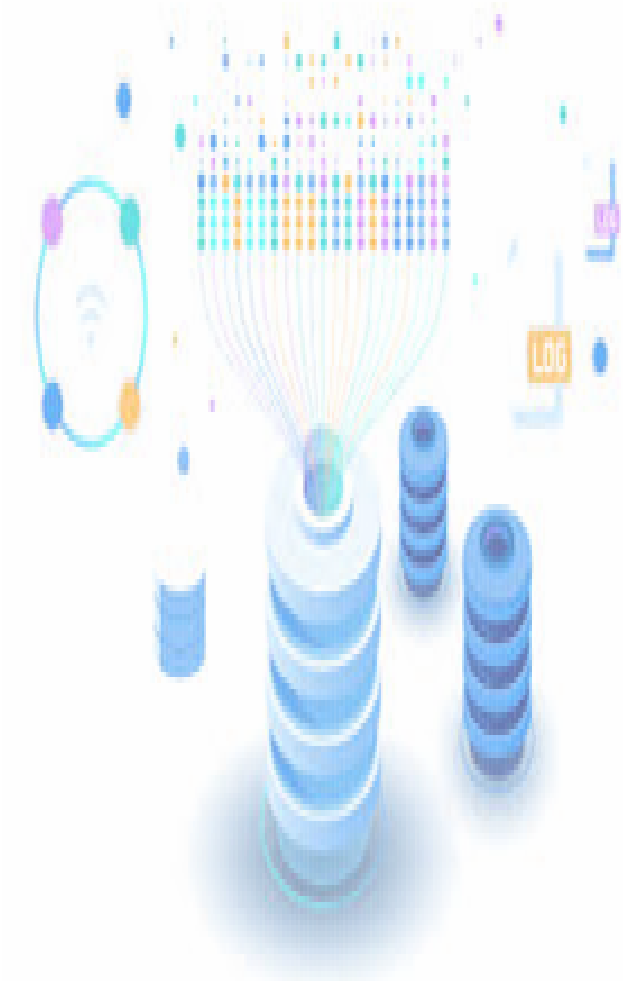


CHAPTER 8

DATABASE PROGRAMMING ON SQL SERVER

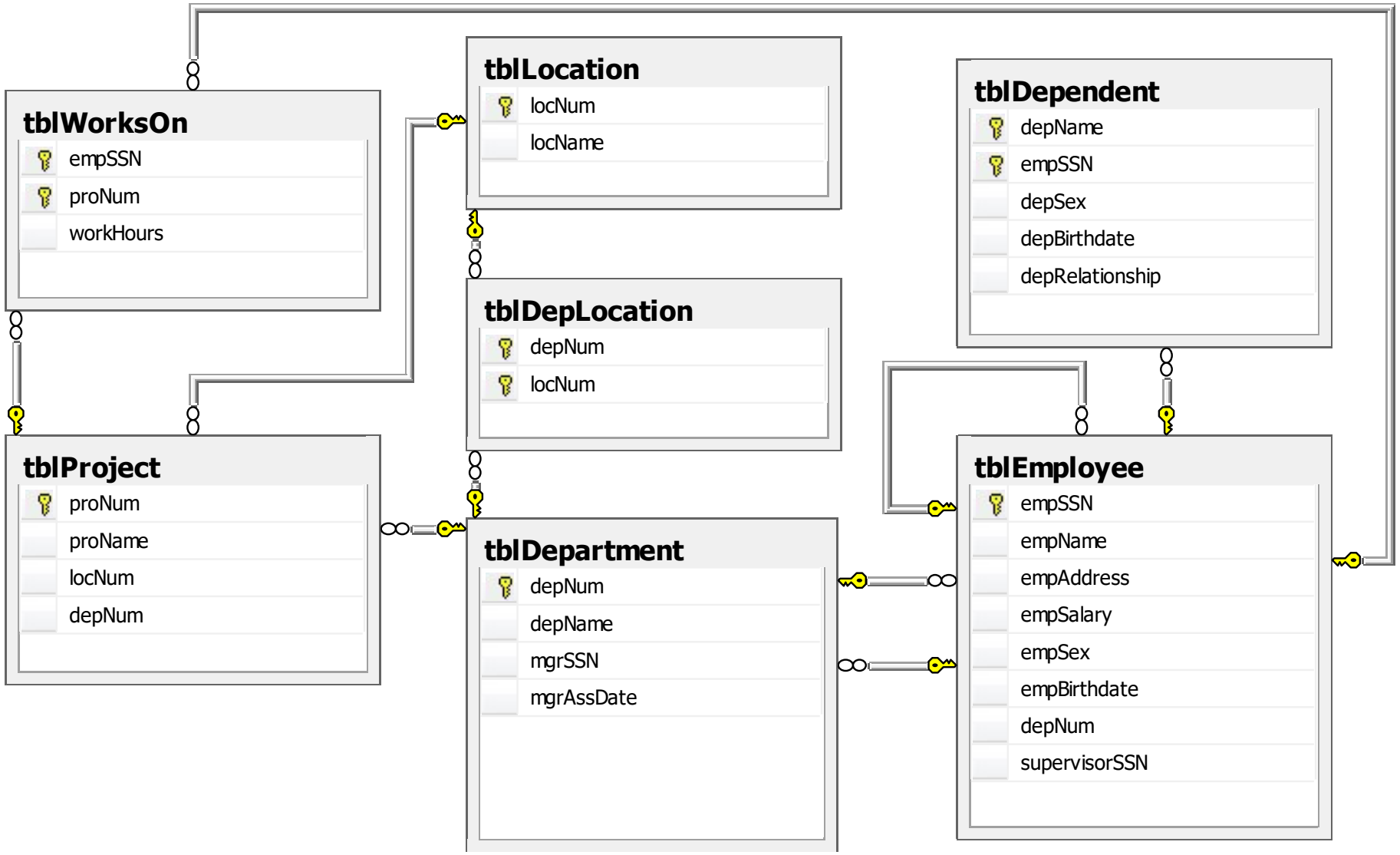
OBJECTIVES

- 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

1) T-SQL Programming
2) Stored-procedure
3) Functions
4) Triggers
5) Cursors

PHYSICAL DIAGRAM - FUHCOMPANY



Variables

There are **2 types of variables**:

- **Global variables:** are variables that are used anywhere in the system. In SQL, global variables are system variables provided by SQL Server.

SQL automatically updates the values for global variables, users cannot assign values directly to these variables.

- **Local variables**: are variables that are only used within the program that declares them.

Local variables start with an @ symbol

Global variables

Global variables	Meaning
@@ERROR	T-SQL statement error number
@@FETCH	STATUS Cursor access status (0 if access status is successful; -1 if unsuccessful)
@@IDENTITY	The identity value added
@@ROWCOUNT	Number of rows in the SQL statement result
@@SERVERNAME	Name of the local server
@@TRANSCOUNT	Number of open transactions
@@VERSION	Information about the SQL Server version in use
@@CURSOR_ROWS	Number of Cursor data rows

2. Local variables

2a) Declare a variable

```
DECLARE
    @variable [AS]  dataType [= value],...
```

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

2b) Assigning Values to Variables

```
SET @VariableName = Value;
```

Variables - Assign a Value

- **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'
```

Variables - Display Value

- 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)
```

2. FLOW-CONTROL STATEMENT

2.1. Statement Blocks:

Begin…End

2.2. Conditional Execution:

IF ... ELSE Statement

CASE ... WHEN

2.3. Looping: WHILE Statement

2.4. Error handling:

@@ERROR

TRY … CATCH

2.1. IF...ELSE (Transact-SQL)

```
IF <condition>
BEGIN
    -- Code to execute if the condition is true
END
ELSE
BEGIN
    -- Code to execute if the condition is false
END
```

Examples:

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
    SELECT 'Weekend';
ELSE
    SELECT 'Weekday';
```

2.2. 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
```

```sql
DECLARE @Salary INT;
SET @Salary = 60000;
IF @Salary > 50000
    BEGIN
        PRINT 'High Salary';
        PRINT 'Eligible for Bonus';
    END
ELSE
    BEGIN
        PRINT 'Low Salary';
        PRINT 'Not Eligible for Bonus';
    END
```

2.2. 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)
SELECT @depNum = A.depNum
FROM tblEmployee A
WHERE A.empName LIKE N'Trần Thiện Bảo'
SET @str=
    CASE @depNum
        WHEN 1 THEN N'Nhóm A'
        WHEN 2 THEN N'Nhóm A'
        ELSE N'Nhóm B'
    END
PRINT @str
```

CASE ... WHEN

```sql
SELECT Name, DepartmentID,
    CASE DepartmentID
        WHEN 1 THEN 'HR'
        WHEN 2 THEN 'Finance'
        WHEN 3 THEN 'IT'
    END AS DepartmentName
FROM Employees;
```

CASE ... WHEN ...with SUM

```
SELECT
    SUM(CASE WHEN DepID=1 THEN Salary ELSE 0 END) AS HR_Sal,
    SUM(CASE WHEN DepID=2 THEN Salary ELSE 0 END) AS FC_Sal
FROM Employees;
```

Key Points:

The CASE expression must always end with **END**.

If no condition matches and there is no ELSE clause, NULL will be returned.

CASE WHEN is evaluated sequentially, meaning the first condition that evaluates to TRUE will return its result

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
```

Handling error using @@ERROR function

The @@ERROR **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
```

HANDLING ERROR USING TRY ... CATCH

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
```

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]
```

```
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
```

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

II.
Stored Procedure

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

```
CREATE PROCEDURE ProcedureName
    @Parameter1 DataType,
    @Parameter2 DataType = DefaultValue
AS
BEGIN
    -- SQL statements go here
END;
```

Key Components:

ProcedureName: The name of the stored procedure.

@Parameter1, @Parameter2: Input parameters (optional) for passing values into the procedure.

AS and BEGIN...END: Indicate the start and end of the procedure logic.

SQL statements: The actual SQL commands (e.g., SELECT, INSERT, UPDATE, etc.) that the procedure will execute.

```sql
CREATE PROCEDURE GetAllEmployees
AS
BEGIN
    SELECT EmployeeID, Name, Salary FROM Employees;
END;
```

```sql
CREATE PROCEDURE GetEmployeeByID
    @EmployeeID INT
AS
BEGIN
    SELECT EmployeeID, Name, Salary
    FROM Employees
    WHERE EmployeeID = @EmployeeID;
END;
```

EXEC GetEmployeeByID @EmployeeID=1001;

Example 1:

A Simple Stored Procedure Without Parameters

```
CREATE PROCEDURE GetAllEmployees
AS
BEGIN
    SELECT EmployeeID, Name, Salary FROM Employees;
END;
```

This stored procedure, GetAllEmployees, when executed, returns all employees from the Employees table.

To Execute:

```
EXEC GetAllEmployees;
```

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
```

3.
Function

3. FUNCTION

In T-SQL, functions are blocks of code that can perform calculations or data manipulation and return results.

There are two main types of functions in T-SQL:

1) Scalar Functions: Return a single value.

2) Table-Valued Functions (TVF): Return a table of data

1. Scalar functions

Scalar functions perform an operation and return a single value like **INT, VARCHAR, DATE,** etc.

```
CREATE FUNCTION FunctionName (@Parameter DataType)
RETURNS DataType
AS
BEGIN
    -- Function logic goes here
    RETURN Result;
END;
```

Creating a function to calculate tax based on salary:

```
CREATE FUNCTION CalculateTax (@Salary DECIMAL(10, 2))
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @Tax DECIMAL(10, 2);
    IF @Salary > 50000
        SET @Tax = @Salary * 0.2;
    ELSE
        SET @Tax = @Salary * 0.1;
    RETURN @Tax;
END;
```

Calling the function:

```
SELECT dbo.CalculateTax(60000) AS TaxAmount;
```

2. Table-Valued Functions (TVF)

Table-valued functions **return a table of data**.

There are two types:

1. **Inline table-valued functions:** Have a single RETURN statement.
2. **Multi-statement table-valued functions:** Can contain multiple statements and variables within the BEGIN...END block.

1. Inline table-valued functions

```
CREATE FUNCTION FunctionName (@Parameter DataType)
RETURNS TABLE
AS
RETURN
(
    SELECT Columns FROM Table WHERE Condition = @Parameter
);
```

1. Inline table-valued functions

Creating a function that returns a list of employees by department:

```
CREATE FUNCTION GetEmployeesByDepartment (@DepartmentID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT EmployeeID, Name, Salary
    FROM Employees
    WHERE DepartmentID = @DepartmentID
);
```

Using the function:

```
SELECT * FROM dbo.GetEmployeesByDepartment(1);
```

2. Multi-statement table-valued functions

Multi-statement TVFs are more complex and can contain multiple statements inside **BEGIN...END.**

```
CREATE FUNCTION FunctionName (@Parameter DataType)
RETURNS @TableVariable TABLE (ColumnDefinitions)
AS
BEGIN
    -- Insert rows into the table variable
    INSERT INTO @TableVariable
    SELECT Columns FROM Table WHERE Condition = @Parameter;

    RETURN;
END;
```

Creating a function that returns employees and their total salaries for departments:

```
CREATE FUNCTION GetEmployeesAndSalaries (@MinSalary DECIMAL(10, 2))
RETURNS @EmployeeData TABLE
(
    EmployeeID INT,
    Name VARCHAR(100),
    Salary DECIMAL(10, 2)
)
AS
BEGIN
    INSERT INTO @EmployeeData
    SELECT EmployeeID, Name, Salary
    FROM Employees
    WHERE Salary >= @MinSalary;

    RETURN;
END;
```

```
SELECT * FROM dbo.GetEmployeesAndSalaries(40000);
```