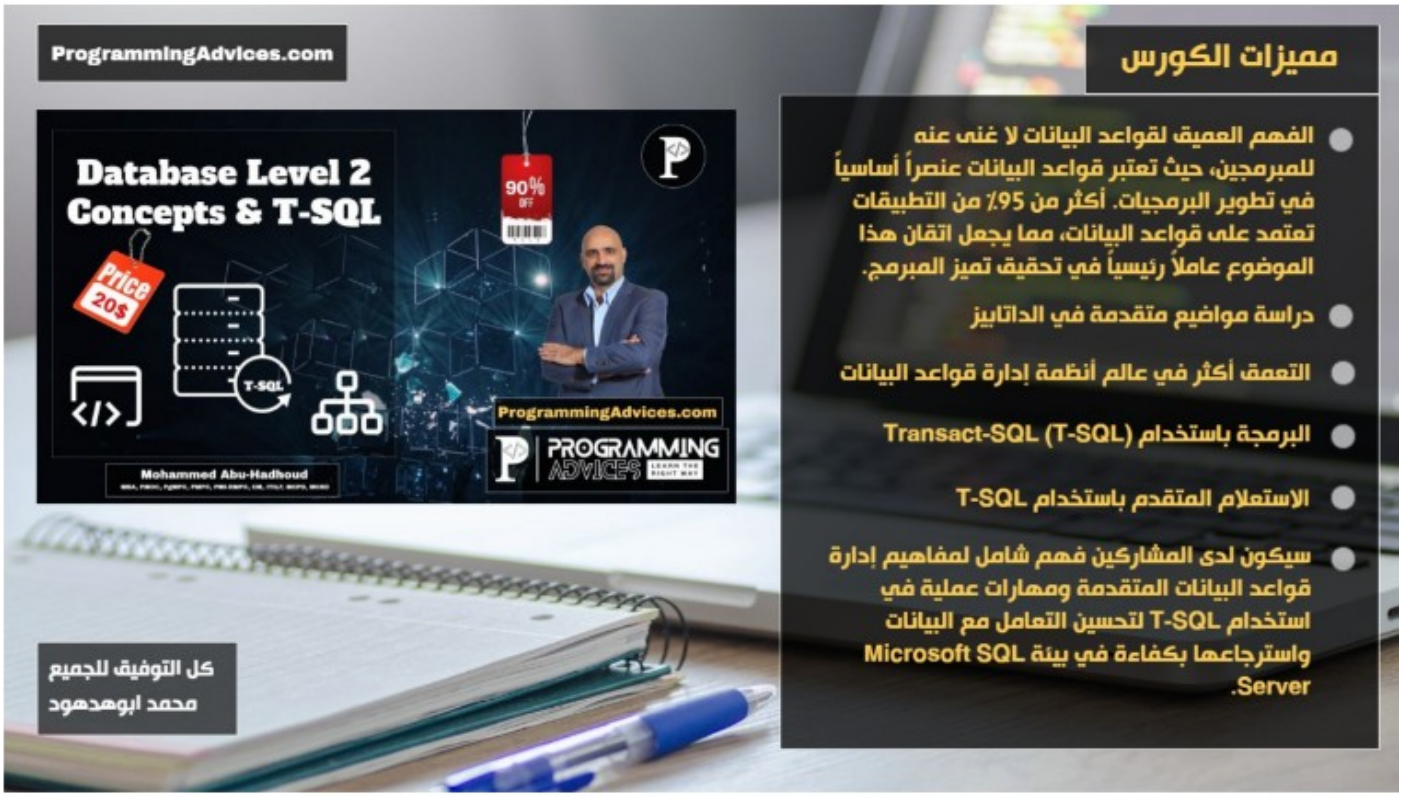
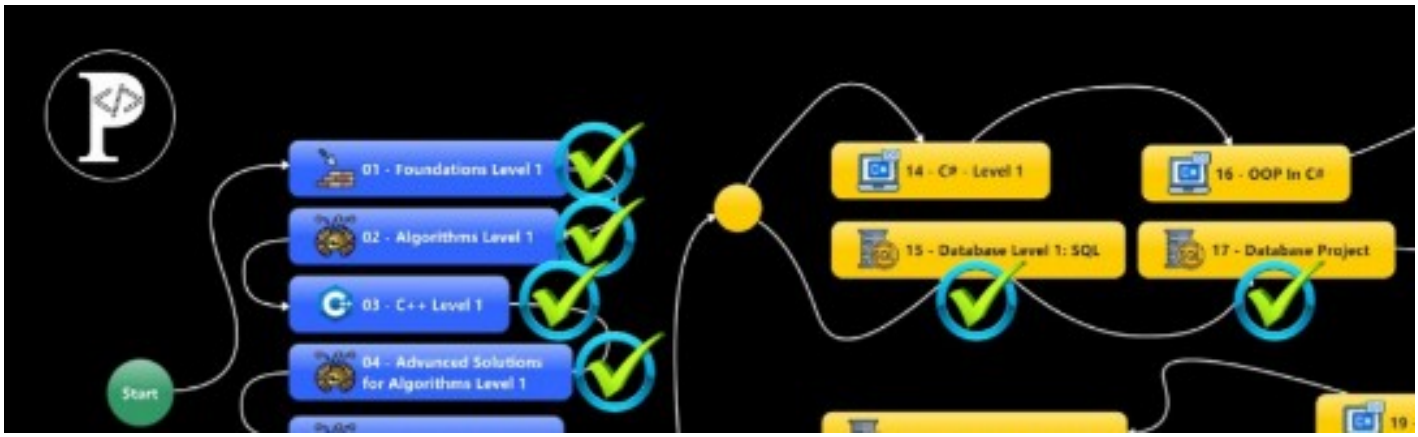
# About This Course




مهم جدا

يجب ان تكون قد انهيت جميع الكورسات المشار اليها بالاخضر




# Telegram Group for This Course


رابط المجموعة على التلجرام

https://t.me/+cFu3yMbJDfU2ZGI0

## Introduction to T-SQL

اول حاجه عايز يعرفهالك هيا انه ال sql هيا query language انما ال t-sql هيا programming language وهيا عباره عن extenstion لل sql

هيا لغة برمجه بسيطه مخصصه للتعامل علي مستوي الداتابيز

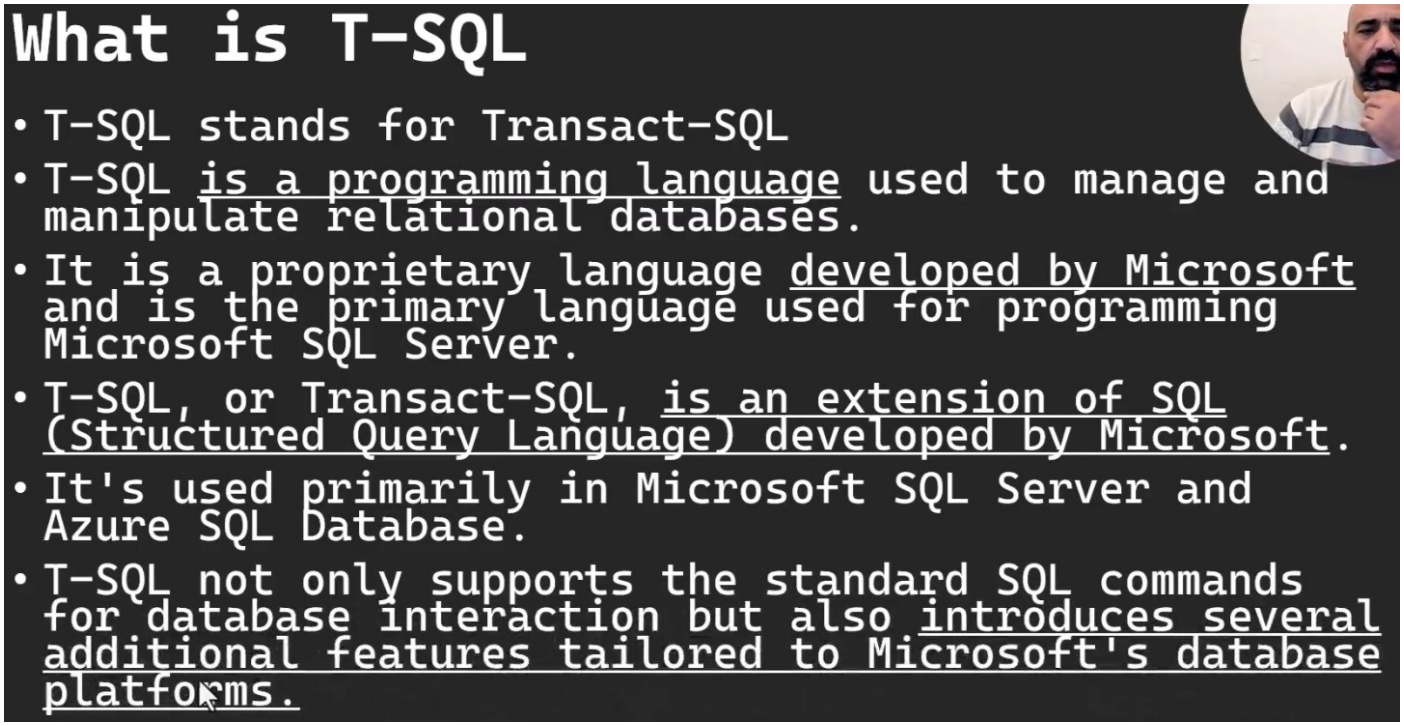
هيا مفيهاش كل الأدوات المتاحه لجميع غات البرمجه لكن فيها حاجات زي ال loops وال functions وأدوات بتمكنك من انك تتعامل مع الداتا بطريقه سريعه

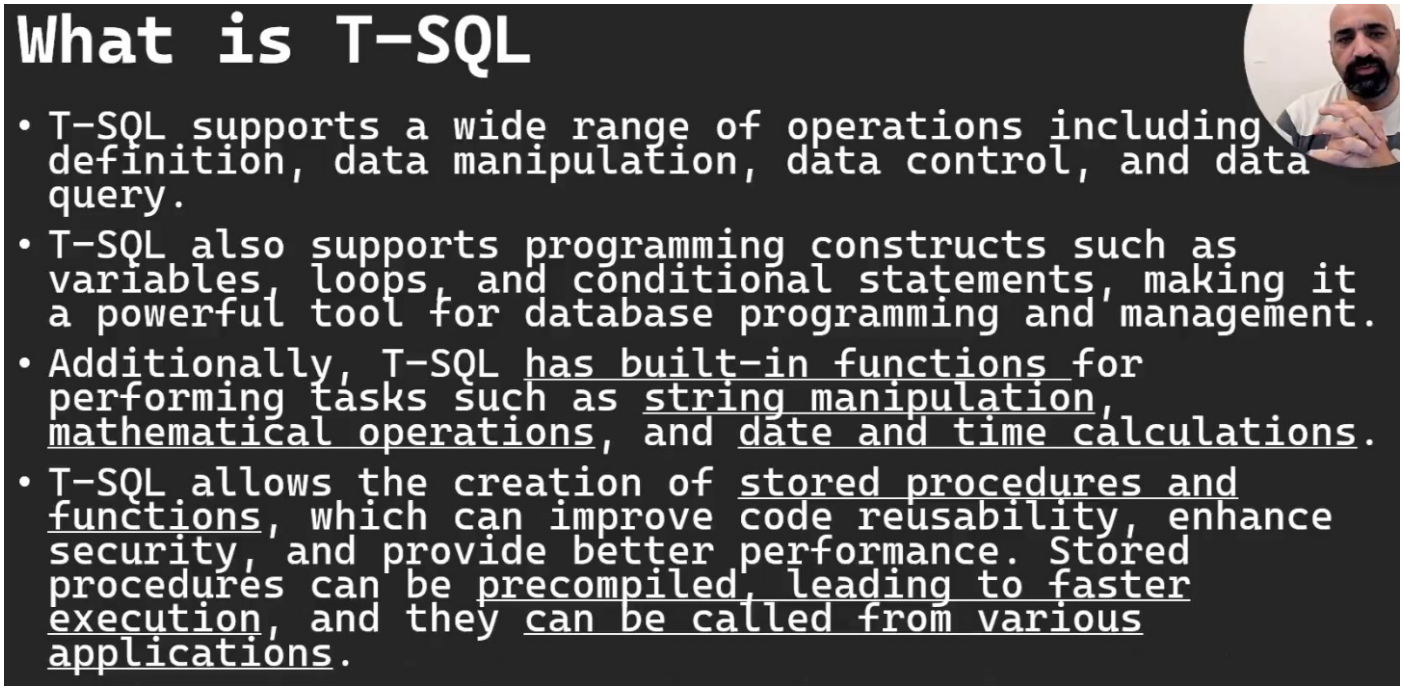
في ال #c لما كنا بنكتب ال queries كنا بنكتبها علي مستوي ال #c انما هنا هنكتبها جوه procedures اسمها stored procedures فهيا بتكون precompiled و optimized وبالتالي ال execution بتاعها بيكون اسرع

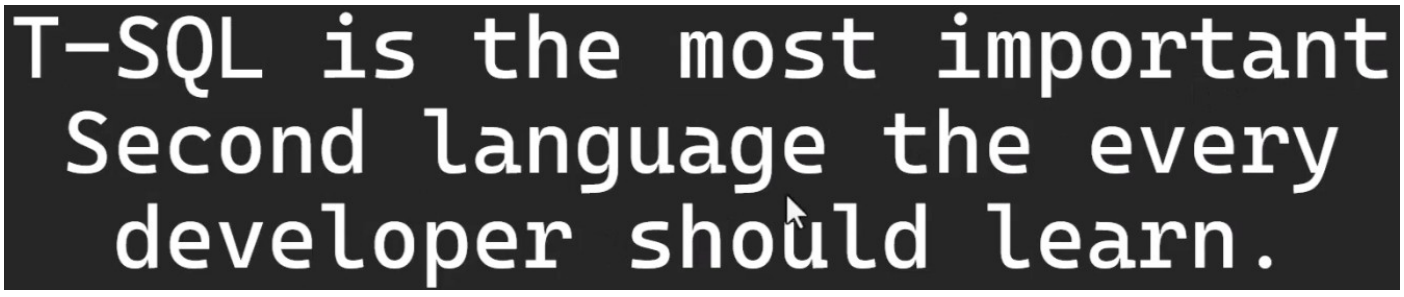
ال t-sql هيا لغة برمجه خاصه بال sql server

في ال oracle عندهم pl-sql نفس المفاهيم بس دي خاصه بال oracle اللي بيفرق هوا ال syntax

وهنا بيقولك انه أي data base مافيهاش لغة برمجه تعتبر ضعيفة




## Introduction to T-SQL

What is T-SQL?

T-SQL, or Transact-SQL, is an extension of SQL (Structured Query Language) developed by Microsoft. It's used primarily in Microsoft SQL Server and Azure SQL Database. T-SQL not only

supports the standard SQL commands for database interaction but also introduces several additional features tailored to Microsoft's database platforms.

- Transact-SQL (T-SQL) is a programming language used to manage and manipulate relational databases. It is a proprietary language developed by Microsoft and is the primary language used for programming Microsoft SQL Server.
- T-SQL is an extension of the SQL (Structured Query Language) standard and adds additional functionality and control over data and database objects. It supports a wide range of operations including data definition, data manipulation, data control, and data query.
- T-SQL also supports programming constructs such as variables, loops, and conditional statements, making it a powerful tool for database programming and management. Additionally, T-SQL has built-in functions for performing tasks such as string manipulation, mathematical operations, and date and time calculations.
- T-SQL allows the creation of stored procedures and functions, which can improve code reusability, enhance security, and provide better performance. Stored procedures can be precompiled, leading to faster execution, and they can be called from various applications.
- T-SQL supports explicit transaction control using keywords like BEGIN TRANSACTION, COMMIT, and ROLLBACK. This helps ensure data integrity by allowing developers to define and manage transaction boundaries explicitly.
- T-SQL provides robust error-handling mechanisms, allowing developers to catch and handle errors gracefully. TRY...CATCH blocks can be used to encapsulate code where errors may occur, making it easier to identify and address issues.
- T-SQL supports both DML and DDL operations, allowing developers to not only query and manipulate data but also define and modify the structure of the database. This comprehensive support streamlines the development and maintenance of database applications.
- T-SQL provides a range of security features, including the ability to define user roles, grant permissions, and implement encryption. This ensures that data is accessed and modified only by authorized users, contributing to a secure database environment.

Overall T-SQL is a powerful and versatile language that is widely used in enterprise environments for managing and manipulating relational databases.

In Oracle there is **PL/SQL**

**T-SQL** (**T**ransact-**SQL**) and **PL/SQL** (**P**rocedural **L**anguage/**SQL**) are both extensions of SQL, the standard language for interacting with relational databases. While they share some similarities, they are distinct in several ways, primarily because they are designed for use with different database management systems.

## Database Compatibility

- T-SQL: Primarily used with **Microsoft SQL Server**.
- PL/SQL: Developed by Oracle Corporation for use with **Oracle Database**.

**Quiz**

## Which of the following best describes T-SQL?

- A data analysis tool
- A user interface design language
- A programming language for managing relational databases
- A hardware configuration language

## What type of language is T-SQL considered to be?

- A markup language
- A scripting language
- A programming language
- A style sheet language

Transact-SQL (T-SQL) provides a robust programming language with features that let you temporarily store values in variables, apply conditional execution of commands, pass parameters to stored procedures, and control the flow of your programs.

- True
- False

## T-SQL is primarily used in which of the following?

- Oracle Database
- MySQL
- Microsoft SQL Server and Azure SQL Database
- MongoDB

## What additional feature does T-SQL offer compared to standard SQL?

- Enhanced graphical interface
- Programming constructs like variables, loops, and conditional statements
- Increased data storage capacity
- Automated database backups

## What are stored procedures in T-SQL used for?

- Increasing data redundancy
- Improving code reusability and performance
- Data encryption
- Creating new database instances

## What kind of operations does T-SQL support?

- Only data query operations
- Data definition, manipulation, control, and query operations
- Only data manipulation operations
- Only data control operations

## Which feature is provided by T-SQL for handling errors?

- Automatic restart
- TRY...CATCH blocks
- Error logging in a separate database
- Email notifications on error

## What distinguishes T-SQL from PL/SQL, which is used in Oracle?

- T-SQL is open-source
- T-SQL is a proprietary language developed by Microsoft
- PL/SQL does not support stored procedures
- T-SQL does not support data manipulation

T-SQL allows the creation of:

- Graphical user interfaces
- Web applications
- Stored procedures and functions
- New database management systems

T-SQL's built-in functions can perform tasks such as:

- Web scraping
- String manipulation and mathematical operations
- Graphic design
- Network configuration

## Variables

طريقة تعريف المتغيرات لنك تكتب كلمة declare وبعدين العلامه @ وبعدها اسم المتغير وبعدين ال data type زي كده

```
DECLARE @EmployeeName VARCHAR(50);
```

عشان تخزن قيمه في المتغير ياما بتستخدم كلمة set ياما كلمة select

```
• Using SET: SET @EmployeeName = 'John Doe';

• Using SELECT: SELECT @EmployeeName = 'John Doe';
```

بعدين تقدر تستخدم المتغيرات في اللي انت عايزه حتي انك تقدر تستخدمها ك parameters في ال query

```
SELECT * FROM Employees WHERE Name =
@EmployeeName;
```

ال scope بتاع المتغيرات بيكون محدود بال batch بتاعه يعني الملف اللي بنكتب فيه ال query او علي مستوي ال stored procedures او علي البرنامج كلها حسب انت هتحطه فين

ال datatypes هيا نفسها ال datatypes بتاعت ال sql

## What is Variable?

- Variables in T-SQL are objects that can hold a single data value of a specific type.
- They are used to store data temporarily during the execution of code.

## Declaring Variables

- Syntax: DECLARE @VariableName DataType;
- Example: DECLARE @EmployeeName VARCHAR(50);
- Here, @EmployeeName is a variable of the type VARCHAR (a string data type) with a maximum length of 50 characters.

## Assigning Values to Variables

You can set a value to a declared variable using the SET or SELECT statement.

- Using SET: SET @EmployeeName = 'John Doe';
- Using SELECT: SELECT @EmployeeName = 'John Doe';

## Using Variables

- Once declared and assigned, you can use variables in your T-SQL code wherever you might use literals or expressions.

- Example in a query:

```
SELECT * FROM Employees WHERE Name =
@EmployeeName;
```

## Variable Scope

- T-SQL variables are local to the batch, stored procedure, or trigger in which they are declared.

- They cease to exist once the batch or procedure completes.

## Data Types

- T-SQL supports various data types for variables, including but not limited to:
  - Integer types (INT, SMALLINT, BIGINT)
  - Decimal types (DECIMAL, NUMERIC)
  - Character types (CHAR, VARCHAR)
  - Date and Time types (DATE, DATETIME)

فيه حاجه اسمها special variable ودي بتستخدم معاها علامة @@ ودي حاجات موجوده built in زي دول

## Special Variables

- @@IDENTITY: Contains the last-inserted identity value.
- @@ROWCOUNT: Contains the number of rows affected by the last statement.

## Best Practices

- Always initialize variables.
- Choose appropriate data types to avoid unnecessary resource consumption.
- Use descriptive names for readability.

## Conclusion

- Variables in T-SQL are essential for writing dynamic and flexible queries.
- They enhance the readability and maintainability of the code by avoiding hard-coded values and allow for more complex logic and operations within SQL scripts and stored procedures.

**Quiz**

What is the primary purpose of variables in T-SQL?

- To permanently store data
- To temporarily hold data during code execution
- To modify the database schema
- To optimize query performance

Which of the following is the correct syntax to declare a variable in T-SQL?

- DECLARE VariableName DataType;
- CREATE @VariableName DataType;
- DECLARE @VariableName DataType;
- SET @VariableName DataType;

How do you assign a value 'John Doe' to a variable @EmployeeName of type VARCHAR?

- SELECT @EmployeeName = 'John Doe';
- SET @EmployeeName = 'John Doe';
- ASSIGN @EmployeeName = 'John Doe';

What is the scope of a T-SQL variable?

- Global to the entire database.
- Local to the batch, stored procedure, or trigger in which it is declared.
- Across multiple databases.
- Within the entire SQL Server instance.

Which special variable in T-SQL holds the last-inserted identity value?

- @@IDENTITY
- @@VARIABLE
- @@LASTID
- @@ROWCOUNT

How can you retrieve the number of rows affected by the last T-SQL statement?

- Using @@ROWCOUNT
- Using @@AFFECTEDROWS
- Using ROW_NUMBER()
- Using COUNT(*)

Which of the following is NOT a valid use of variables in T-SQL?

- Storing the result of a query
- Using in conditional logic in stored procedures
- Replacing the name of a table or column in a SQL statement
- Temporarily holding data for processing

What happens to a local variable after the batch or stored procedure in which it is declared completes?

- It is stored in the database.
- It remains accessible until the database connection is closed.
- It is automatically deleted or de-allocated.
- It becomes a global variable.

## Example 1 - Employee Report

الملف ده هتفتحه وتنفذه عشان تعمل داتابيز جديده


ده الكود اللي جواه

```
Create Database C21_DB1;
Go

Use C21_DB1;


-- Create the Departments Table
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    Name VARCHAR(50)
);

-- Insert sample data into Departments
INSERT INTO Departments (DepartmentID, Name) VALUES (1, 'Human Resources');
INSERT INTO Departments (DepartmentID, Name) VALUES (2, 'Marketing');
INSERT INTO Departments (DepartmentID, Name) VALUES (3, 'Sales');
INSERT INTO Departments (DepartmentID, Name) VALUES (4, 'IT');

Go

--Create the Employees Table
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50),
    DepartmentID INT,
    HireDate DATE,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);

-- Insert sample data into Employees
INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (1, 'John Smith', 3, '2023-
01-10');
INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (2, 'Jane Doe', 3, '2023-02-
```

```
15');
INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (3, 'Emily Davis', 2, '2023-
03-20');
INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (4, 'Michael Brown', 1,
'2022-11-05');
INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (5, 'Sarah Miller', 4,
'2023-01-05');
```

الملف ده في الكود بتاع البرنامج اللي هنعمله بال tsql

02 Example 1 - Employee Report

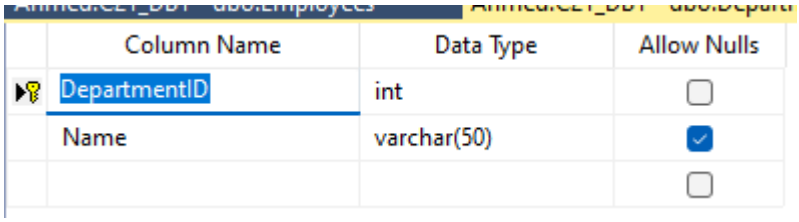
طيب الكود ده عباره عن ايه؟

تعالي نمسكه واحده واحده

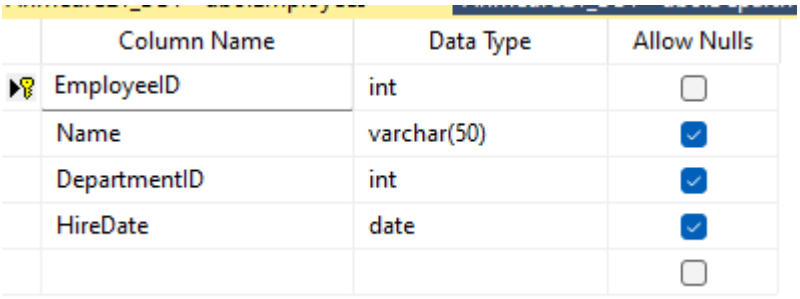
احنا دلوقتي عندنا الداتا بيز مكونه من جدولين

او جدول هوا ال departments وده فيه اسم القسم وال id بتاعه

والجدول التاني فيه اسامي الموظفين والاقسام التابعين ليها وتاريخ التعيين



| Column Name | Data Type | Allow Nulls |
|---|---|---|
| DepartmentID | int | ☐ |
| Name | varchar(50) | ☑ |
| | | ☐ |



| Column Name | Data Type | Allow Nulls |
|---|---|---|
| EmployeeID | int | ☐ |
| Name | varchar(50) | ☑ |
| DepartmentID | int | ☑ |
| HireDate | date | ☑ |
| | | ☐ |

طيب لو جيت قولتلك عايز برنامج اديله رقم القسم واحددله مدة زمنية معينه يطبعلي اسم القسم والمده اللي انا اديتهاله وعدد الموظفين اللي اتعينوا في الفتره دي؟

طيب تعالي الأول نعملها بال query

```
select Departments.Name,COUNT(*)
from Departments inner join Employees
on Departments.DepartmentID=Employees.DepartmentID
where Departments.DepartmentID=3
and( Employees.HireDate between cast('2023-01-01' as Date)
```

```
and cast('2023-12-31' as Date))
group by Departments.Name
```

طيب لو عملناها كبرنامج هتبقي عامله ازاي؟

اول حاجه محتاجين متغيرات نخزن فيها القيم سواء اللي هندخلها ك parameters او هنخزن فيها الداتا اللي خارجه من الداتا بيز

طيب احنا هندخل ال id بتاع القسم وتاريخ البدايه وتاريخ النهايه

والداتا اللي هتطلع هتكون اسم القسم وعدد الموظفين

```
use C21_DB1;
declare @DepartmentID INT;
declare @DepartmentName nvarchar(50);
declare @StartDate DATE;
declare @EndDate DATE;
declare @TotalEmployees INT;
```

تاني حاجه هنخزن القيم اللي عايزن نطلع الداتا علي أساسها جوه المتغيرات

```
set @StartDate = '2023-01-01';
set @EndDate = '2023-12-31';
set @DepartmentID=3;
```

كده فاضل المتغيرات الخاصه بالنتايج ودي هنخزن فيها الداتا عن طريق ال select statement

فاول حاجه هنخزن اسم القسم

```
select @DepartmentName=Departments.Name
from Departments
where Departments.DepartmentID=@DepartmentID;
```

وتاني حاجه هنخزن عدد الموظفين

```
select @TotalEmployees=COUNT(*)
from Employees
where Employees.DepartmentID=@DepartmentID
AND (Employees.HireDate between @StartDate And @EndDate);
```

كده مش فاضل غير اننا نطبع التقرير عالشاشه

```
PRINT 'Department Report';
```

```
PRINT 'Department Name: ' + @DepartmentName;
PRINT 'Reporting Period: ' + CAST(@StartDate AS VARCHAR) +
' to ' + CAST(@EndDate AS VARCHAR);
PRINT 'Total Employees Hired in ' + CAST(YEAR(@StartDate)
AS VARCHAR) + ': ' + CAST(@TotalEmployees AS VARCHAR);
```

وده الكود كله

```
use C21_DB1;

-- Example: Employee Report Generation in T-SQL
-- This script demonstrates the declaration,
initialization, and use of variables in T-SQL.
-- It generates a report for a specific department,
including the department name, reporting period, and total
employees hired within that period.
-- This comprehensive script gives a practical insight into
how variables can be effectively used in T-SQL to create
dynamic and flexible SQL scripts.


-- Step 1: Declare variables
DECLARE @DepartmentID INT; -- Variable for department ID
DECLARE @StartDate DATE; -- Variable for start date
DECLARE @EndDate DATE; -- Variable for end date
DECLARE @TotalEmployees INT; -- Variable to hold the total
number of employees
DECLARE @DepartmentName VARCHAR(50); -- Variable for
department name

-- Step 2: Initialize variables
SET @DepartmentID = 3; -- Assign a specific department ID
SET @StartDate = '2023-01-01'; -- Set start date for the
report
SET @EndDate = '2023-12-31'; -- Set end date for the report

-- Step 3: Retrieve department name based on department ID
SELECT @DepartmentName = Name FROM Departments WHERE
DepartmentID = @DepartmentID;
```

```sql
-- Step 4: Calculate the total number of employees in the
specified department
SELECT @TotalEmployees = COUNT(*)
FROM Employees
WHERE DepartmentID = @DepartmentID
AND HireDate BETWEEN @StartDate AND @EndDate;

-- Step 5: Print the report
PRINT 'Department Report';
PRINT 'Department Name: ' + @DepartmentName;
PRINT 'Reporting Period: ' + CAST(@StartDate AS VARCHAR) +
' to ' + CAST(@EndDate AS VARCHAR);
PRINT 'Total Employees Hired in ' + CAST(YEAR(@StartDate)
AS VARCHAR) + ': ' + CAST(@TotalEmployees AS VARCHAR);

select Departments.Name,COUNT(*)
from Departments inner join Employees
on Departments.DepartmentID=Employees.DepartmentID
where Departments.DepartmentID=3
and( Employees.HireDate between cast('2023-01-01' as Date)
and cast('2023-12-31' as Date))
group by Departments.
```

## Example 2 - Monthly Sales Summary Report

شغل الكود ده عشان يعمل جدول جديد في نفس الداتا بيز اللي فاتت

```sql
use C21_DB1;

-- Insert sample data into Sales
CREATE TABLE Sales (
   SaleID INT PRIMARY KEY,
   SaleDate DATE,
   SaleAmount DECIMAL(10, 2)
);

-- Inserting sample data into Sales
INSERT INTO Sales (SaleID, SaleDate, SaleAmount) VALUES (1,
'2023-06-01', 150.00);
INSERT INTO Sales (SaleID, SaleDate, SaleAmount) VALUES (2,
'2023-06-03', 200.00);
```

```
INSERT INTO Sales (SaleID, SaleDate, SaleAmount) VALUES (3,
'2023-06-05', 50.00);
INSERT INTO Sales (SaleID, SaleDate, SaleAmount) VALUES (4,
'2023-07-10', 300.00);
INSERT INTO Sales (SaleID, SaleDate, SaleAmount) VALUES (5,
'2023-07-11', 250.00);
```

طيب دلوقتي احنا عايزين نعمل ايه؟

قالك احنا عاوزين برنامج نديله الشهر والسنه منفصلين يطبعلنا اجمالي المبيعات وعدد عمليات البيع اللي تمت ومتوسط المبيعات خلال الشهر اللي دخلناه ده

اول حاجه هنعملها هنعرف المتغيرات اللي هنخزن فيها الداتا

```
use C21_DB1

declare @Year INT
declare @Month INT
declare @TotalSales decimal(10,2)
declare @TotalTransactions INT
declare @AverageSale decimal(10,2)
```

بعدين ندخل القيم

```
set @Year=2023
set @Month=6
```

بعدين نجيب الداتا من الداتابيز

```
select @TotalSales=SUM(Sales.SaleAmount)
from Sales
where YEAR(Sales.SaleDate)=@Year
and MONTH(Sales.SaleDate)=@Month

select @TotalTransactions= COUNT(*)
from Sales
where YEAR(Sales.SaleDate)=@Year
and MONTH(Sales.SaleDate)=@Month

SET @AverageSale=@TotalSales/@TotalTransactions
```

والباقي جملة الطباعه

وده الكود كله

```
/*
In this scenario, we'll use T-SQL variables to generate a monthly sales summary report for a given
year and month.
This report will include total sales, number of transactions, and average sale value.
We'll need a Sales table that contains details of each sale.
*/

-- This script generates a monthly sales summary report.
-- It includes total sales, total number of transactions, and the average sale value for a specified
month and year.

-- Declare variables
DECLARE @Year INT;
DECLARE @Month INT;
DECLARE @TotalSales DECIMAL(10, 2);
DECLARE @TotalTransactions INT;
DECLARE @AverageSale DECIMAL(10, 2);

-- Initialize variables
SET @Year = 2023; -- Set the year for the report
SET @Month = 6; -- Set the month for the report

-- Calculate total sales for the specified month and year
SELECT @TotalSales = SUM(SaleAmount)
FROM Sales
WHERE YEAR(SaleDate) = @Year AND MONTH(SaleDate) = @Month;

-- Calculate the total number of transactions
SELECT @TotalTransactions = COUNT(*)
FROM Sales
WHERE YEAR(SaleDate) = @Year AND MONTH(SaleDate) = @Month;

-- Calculate the average sale value
SET @AverageSale = @TotalSales / @TotalTransactions;

-- Print the report
PRINT 'Monthly Sales Summary Report';
PRINT 'Year: ' + CAST(@Year AS VARCHAR) + ', Month: ' + CAST(@Month AS VARCHAR);
PRINT 'Total Sales: ' + CAST(@TotalSales AS VARCHAR);
PRINT 'Total Transactions: ' + CAST(@TotalTransactions AS VARCHAR);
PRINT 'Average Sale Value: ' + CAST(@AverageSale AS VARCHAR);
```

## Example 3 - Employee Attendance Tracking

ده الكود بتاع الجدول

```
use C21_DB1;


-- Inserting sample data into EmployeeAttendance
CREATE TABLE EmployeeAttendance (
    RecordID INT PRIMARY KEY,
    EmployeeID INT,
    AttendanceDate DATE,
    Status VARCHAR(10) -- Values could be 'Present', 'Absent', 'Leave'
);

-- Inserting sample data into EmployeeAttendance
INSERT INTO EmployeeAttendance (RecordID, EmployeeID, AttendanceDate, Status) VALUES (1, 101, '2023-
07-01', 'Present');
```

```
INSERT INTO EmployeeAttendance (RecordID, EmployeeID, AttendanceDate, Status) VALUES (2, 102, '2023-07-01', 'Absent');
INSERT INTO EmployeeAttendance (RecordID, EmployeeID, AttendanceDate, Status) VALUES (3, 103, '2023-07-01', 'Leave');
INSERT INTO EmployeeAttendance (RecordID, EmployeeID, AttendanceDate, Status) VALUES (4, 101, '2023-07-02', 'Present');
INSERT INTO EmployeeAttendance (RecordID, EmployeeID, AttendanceDate, Status) VALUES (5, 102, '2023-07-02', 'Present');
-- Add more data as needed
```

المرادي عاوزين ندخل رقم الموظف ورقم السنه ورقم الشهر يجيبلنا عدد الأيام في الشهر وعدد أيام الحضور وعدد أيام الغياب وعدد الأيام اللي مشي فيها بدري

كالعاده نبدأ بالمتغيرات

```
declare @ReportYear int
declare @ReportMonth int
declare @TotalDayes int
declare @EmployeeID int
declare @PresentDays int
declare @AbsentDays int
declare @LeaveDays int
```

ونعين القيم اللي عايزين ندخلها

```
set @ReportYear=2023
set @ReportMonth=7
set @EmployeeID=101
```

ونجيب الداتا من الداتابيز

هيا جمله واحده وهنغير بس ال status

```
select @AbsentDays= count(*)
from EmployeeAttendance
where EmployeeAttendance.EmployeeID=@EmployeeID
and YEAR(EmployeeAttendance.AttendanceDate)=@ReportYear
and MONTH(EmployeeAttendance.AttendanceDate)=@ReportMonth
and EmployeeAttendance.Status='Absent'
```

وده الكود كله

```
declare @ReportYear int
declare @ReportMonth int
declare @TotalDayes int
declare @EmployeeID int
declare @PresentDays int
```

```sql
declare @AbsentDays int
declare @LeaveDays int

set @ReportYear=2023
set @ReportMonth=7
set @EmployeeID=101

set @TotalDayes=DAY(EOMONTH(DATEFROMPARTS(@ReportYear,@ReportMonth,1)))

select @PresentDays= count(*)
from EmployeeAttendance
where EmployeeAttendance.EmployeeID=@EmployeeID
and YEAR(EmployeeAttendance.AttendanceDate)=@ReportYear
and MONTH(EmployeeAttendance.AttendanceDate)=@ReportMonth
and EmployeeAttendance.Status='Present'

select @AbsentDays= count(*)
from EmployeeAttendance
where EmployeeAttendance.EmployeeID=@EmployeeID
and YEAR(EmployeeAttendance.AttendanceDate)=@ReportYear
and MONTH(EmployeeAttendance.AttendanceDate)=@ReportMonth
and EmployeeAttendance.Status='Absent'

select @LeaveDays= count(*)
from EmployeeAttendance
where EmployeeAttendance.EmployeeID=@EmployeeID
and YEAR(EmployeeAttendance.AttendanceDate)=@ReportYear
and MONTH(EmployeeAttendance.AttendanceDate)=@ReportMonth
and EmployeeAttendance.Status='Leave'

print 'Employee Attendance report of Employee: '+cast( @EmployeeID as nvarchar)
print 'Report Month: '+cast(@ReportMonth as nvarchar)+'/'+cast(@ReportYear as nvarchar)
print 'Total working days: '+cast(@TotalDayes as nvarchar)
print 'Total Present days: '+cast(@PresentDays as nvarchar)
print 'Total Absent days: '+cast(@AbsentDays as nvarchar)
print 'Total Leave days: '+cast(@LeaveDays as nvarchar)
```

## Example 4 - Loyalty points

ده الجدول اللي هنزوده

```sql
Use C21_DB1;

-- Creating Customers table
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(50),
    LoyaltyPoints INT
);

-- Inserting sample data into Customers
INSERT INTO Customers (CustomerID, Name, LoyaltyPoints) VALUES (1, 'John Doe', 0);
INSERT INTO Customers (CustomerID, Name, LoyaltyPoints) VALUES (2, 'Jane Smith', 0);
INSERT INTO Customers (CustomerID, Name, LoyaltyPoints) VALUES (3, 'Emily White', 0);

-- Creating Purchases table
CREATE TABLE Purchases (
    PurchaseID INT PRIMARY KEY,
    CustomerID INT,
    PurchaseDate DATE,
    Amount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Inserting sample data into Purchases
INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate, Amount) VALUES (1, 1, '2023-07-01',
```

```
120.00);
INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate, Amount) VALUES (2, 2, '2023-07-02',
60.00);
INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate, Amount) VALUES (3, 1, '2023-07-03',
30.00);
-- Add more data as needed
```

بالكود اللي فوق ده احنا عملنا جدولين جدول للعملاء وجدول للمشتريات بتاعتهم

دلوقتي عايزين ندخل رقم عميل نجيب الفلوس اللي صرفها عندنا ونقسمها عل 10 تدينا النقط اللي هنزودهاله

وبعدين نروح نزودله النقط دي في الجدول بتاع العملاء

نحدد المتغيرات الأول

```
declare @CustomerID int
declare @CurrentYear int
declare @TotalSpent decimal(10,2)
declare @PointsEarned int
```

وبعدين نعين القيم

```
set @CustomerID=1
set @CurrentYear=YEAR(GETDATE())

select @TotalSpent=sum(Purchases.Amount) from Purchases
where Purchases.CustomerID= @CustomerID
and YEAR(Purchases.PurchaseDate)=@CurrentYear

SET @PointsEarned=(@TotalSpent/10)
```

كده فاضل نضيف النقط للعميل

```
UPDATE Customers
SET LoyaltyPoints=@PointsEarned
WHERE Customers.CustomerID=@CustomerID
```

والباقي جمل طباعه

وده الكود كله

```
/*
In this scenario, we will use T-SQL variables to calculate and update loyalty points for customers
based on their purchase history.
The calculation will be based on the total amount spent by the customer in a given year,
with a simple point system where 1 point is awarded for every $10 spent.
```

```
This example demonstrates how T-SQL variables can be used for more complex calculations
involving data from multiple tables, and how these results can be used to update records in a
database,
showcasing the power and versatility of SQL in handling real-world business scenarios.

*/
-- Declare variables
DECLARE @CustomerID INT;
DECLARE @TotalSpent DECIMAL(10, 2);
DECLARE @PointsEarned INT;
DECLARE @CurrentYear INT = YEAR(GETDATE());

-- Initialize CustomerID
SET @CustomerID = 1; -- Example: Calculate points for CustomerID 1

-- Calculate total amount spent by the customer in the current year
SELECT @TotalSpent = SUM(Amount)
FROM Purchases
WHERE CustomerID = @CustomerID AND YEAR(PurchaseDate) = @CurrentYear;

-- Calculate loyalty points (1 point for every $10 spent)
SET @PointsEarned = CAST(@TotalSpent / 10 AS INT);

-- Update loyalty points in Customers table
UPDATE Customers
SET LoyaltyPoints = LoyaltyPoints + @PointsEarned
WHERE CustomerID = @CustomerID;

-- Print the results
PRINT 'Loyalty Points Update for Customer ID: ' + CAST(@CustomerID AS VARCHAR);
PRINT 'Total Amount Spent in ' + CAST(@CurrentYear AS VARCHAR) + ': $' + CAST(@TotalSpent AS
VARCHAR);
PRINT 'Loyalty Points Earned: ' + CAST(@PointsEarned AS VARCHAR);

-- This script calculates and updates the loyalty points for a customer based on their total
spending in the current year.
```

## Simple IF Statement

هيا نفسها ال if statement بتاعت أي لغة برمجه بس فيه اختلافين

اول اختلاف انك بتحط = يساوي واحده مش اتنين

تاني اختلاف انك بدل ماتفتح اقواس لا بتكتب begin و end

```
-- you can use >, < , =, != , >= , <=
```

# IF Statement in T-SQL

Introduction to IF Statement in T-SQL

The IF statement in T-SQL is a control-of-flow language construct that allows you to execute or skip a statement block based on a specified condition. It is akin to "if-then" logic found in many programming languages.

**Syntax of IF Statement**

The basic syntax of an IF statement in T-SQL is:

```
IF condition
BEGIN
    -- Statements to execute if the condition is true
END
ELSE
BEGIN
    -- Statements to execute if the condition is false
END
```

- `condition`: A boolean expression that evaluates to true or false.
- `BEGIN` and `END`: Define the start and end of a block of statements.

**Simple IF Statement**

Without the ELSE part, the IF statement executes the block only if the condition is true.

```
Declare @a int, @b int;
set @a = 20;
set @b=10;

IF @a > @b
        BEGIN
                PRINT 'Yes A is greater than B'
        END
```

## IF...ELSE Statement

مش محتاجه شرح

## IF .. Else Statement in T-SQL

IF...ELSE Statement

Including ELSE allows an alternate action if the condition is false.

```
Declare @year int;
set @year =2001;

IF @year >= 2000
```

```
    BEGIN
        PRINT '21st century'
    END
ELSE
    BEGIN
        PRINT '20th century or earlier'
    END
```

## Nested IF Statements

### Nested IF Statements

You can nest IF statements within each other for complex conditions.

```
Declare  @score int;
set @score = 92;

IF @score >= 90
        BEGIN
                PRINT 'Grade A'
        END
ELSE
        BEGIN
                IF @score >= 80
                        BEGIN
                                PRINT 'Grade B'
                        END
                ELSE
                        BEGIN
                                PRINT 'Grade C or lower'
                        END
        END
```

## Using IF with Variables and Conditional Assignments

### Using IF with Variables

Variables can be used within an IF statement for dynamic conditions.

```
DECLARE @age INT;
```

```
SET @age = 25;



IF @age >= 18
BEGIN
    PRINT 'Adult'
END
ELSE
BEGIN
    PRINT 'Minor'
END
```

## Conditional Assignment

IF statements are often used for conditional assignment to variables.

```
DECLARE @max INT;
Declare @a int, @b int;
set @a = 20;
set @b=10;



IF @a > @b
    SET @max = @a
ELSE
    SET @max = @b

Print @max;
```

**Using IF Statement with AND/OR/NOT**

```
DECLARE @Age INT = 25;
DECLARE @Salary DECIMAL(10,2) = 50000;

IF (@Age > 18 AND @Salary >= 50000)
    BEGIN
        PRINT 'Eligible for the loan';
    END
```

```
ELSE
    BEGIN
        PRINT 'Not eligible for the loan';
    END

------------------------------

DECLARE @Grade CHAR(1) = 'B';
DECLARE @AttendancePercentage INT = 75;

IF @Grade = 'A' OR @AttendancePercentage > 70
    BEGIN
        PRINT 'Qualified for extra-curricular activities';
    END
ELSE
    BEGIN
        PRINT 'Not qualified for extra-curricular
activities';
    END

--------------------------

DECLARE @CustomerStatus NVARCHAR(10) = 'Inactive';

IF NOT (@CustomerStatus = 'Active')
    BEGIN
        PRINT 'Send re-engagement email';
    END
ELSE
    BEGIN
        PRINT 'Customer is active';
    END
```

## Error Handling with IF

في ال t sql لو حصل خطأ معين رقم الخطأ ده بيرجع في متغير اسمه @@ERROR ولو مفيش أخطاء بيرجع صفر

```
--Error Handling with IF

DECLARE @ErrorValue INT;
```

```sql
-- Example SQL operation
INSERT INTO Employees (Name) VALUES ('John Doe');

-- Capture the error immediately
SET @ErrorValue = @@ERROR;


-- Check and respond to the error
IF @ErrorValue <> 0
BEGIN
   PRINT 'An error occurred with error number: ' +
CAST(@ErrorValue AS VARCHAR);
   -- Additional error handling logic
END
```

## Error Handling with IF

IF statements can be used to handle errors or unexpected conditions.

```
IF @@ERROR <> 0
BEGIN
    PRINT 'An error occurred.'
END
```

## Key Characteristics of `@@ERROR`

1. Value: After each Transact-SQL statement, `@@ERROR` holds the error number of that statement. If the statement executed successfully, `@@ERROR` returns 0.
2. Reset After Each Statement: `@@ERROR` is reset to 0 after each T-SQL statement, regardless of whether an error occurred. This means you must check `@@ERROR` immediately after the statement that might have caused an error.
3. Usage: It's often used in conjunction with the `IF` statement to check for errors and take appropriate action.

## Limitations and Best Practices

1. Immediate Check: Always check `@@ERROR` immediately after the statement you're interested in, because its value is reset after each SQL statement.

2. Superseded by TRY...CATCH: In modern T-SQL programming, `@@ERROR` is often replaced by the more robust TRY...CATCH construct, which offers better error handling capabilities.
3. Not Always Reliable: `@@ERROR` might not catch all types of errors, especially those that are severe enough to terminate the connection.
4. Use in Transactions: In transaction processing, use `@@ERROR` to decide whether to commit or roll back a transaction.

## Conclusion

While `@@ERROR` provides a basic mechanism for error checking in T-SQL, its limitations mean it's often better to use the TRY...CATCH block for more comprehensive error handling. Understanding `@@ERROR` is still useful, particularly for maintaining and understanding legacy SQL Server code.

## Using IF with EXISTS

اخدنا ال EXISTS قبل كده وقولنا انها بتاخد QUERY ولو ال QUERY رجعتلي ولو RECORD واحد ال EXISTS هترجعلي TRUE ولو كده هترجع FALSE

هنا بقي بيحطهالك مع ال IF STATEMENT

## Using IF with EXISTS

IF statements can be combined with EXISTS to check for the existence of rows in a table that meet a certain condition.

```
IF EXISTS (SELECT * FROM Employees WHERE Name = 'John Smith')
        BEGIN
                PRINT 'Yes, John Smith is there.'
        END
ELSE
        BEGIN
                PRINT 'No, John Smith is there.'
        END
```

## Best Practices & Summary

## Best Practices

- Keep the logic within IF statements simple for better readability.
- Avoid deeply nested IF statements when possible. Consider alternative structures like CASE statements or stored procedures.
- Ensure conditions are clear and cover all expected cases.

## Summary

The IF statement in T-SQL is a fundamental tool for controlling the flow of execution based on conditions. It's versatile and can be used in a variety of scenarios, from simple checks to complex decision-making processes. Understanding and utilizing IF statements effectively can greatly enhance the functionality and efficiency of your SQL scripts.

## Quiz

What is the purpose of the IF statement in T-SQL?

- To execute or skip a statement block based on a specified condition
- To create a boolean expression in T-SQL
- To handle errors or unexpected conditions in T-SQL
- To declare variables in T-SQL

What is the syntax of an IF statement in T-SQL?

- IF condition, BEGIN, END
- IF condition; BEGIN; END
- IF condition: BEGIN; END
- IF condition BEGIN, END

## What happens if an IF statement does not have an ELSE part?

- The IF statement will not execute
- The IF statement will execute if the condition is true
- The IF statement will execute if the condition is false
- The IF statement will always execute

## Can IF statements be nested in T-SQL?

- No, IF statements cannot be nested
- Yes, IF statements can be nested
- Only two levels of nesting are allowed
- Nested IF statements can only be used with variables

## What can variables be used for within an IF statement?

- To check for the existence of specific rows in a table
- To handle errors or unexpected conditions
- To execute dynamic conditions
- To assign a value based on a condition

## What is the purpose of @@ERROR in T-SQL?

- To execute or skip a statement block based on a specified condition
- To handle errors or unexpected conditions in T-SQL
- To declare variables in T-SQL
- To check for the existence of specific rows in a table

## How can you create an alternate action for when an IF statement's condition is false in T-SQL?

- Use ELSE
- Use OTHERWISE
- Use EXCEPT
- No alternate action is possible

## What does a nested IF statement allow you to do?

- Run the same code multiple times.
- Check for multiple conditions in a sequence.
- Repeat a block of code until a condition is met.
- Execute a block of code asynchronously.

What does the IF EXISTS clause check in T-SQL?

- The existence of rows in a table that meet a certain condition.
- The existence of a database.
- Whether a variable exists or not.
- If a table exists in the database.

In error handling, what does IF @@ERROR <> 0 signify in T-SQL?

- There were no errors in the previous SQL statement.
- Resets the error status.
- Checks if the last SQL statement caused an error.
- Compares two error messages.

## CASE Statement

هنا بيقولك انه في ال T SQL مفيش حاجه مقابله لل switch

البديل هوا ال case بس دي بتشتغل في ال query يعني ماتقدرش تحطها في كود عادي

# CASE Statement in T-SQL

### Introduction

T-SQL does not have a dedicated `SWITCH` statement as found in many programming languages. Instead, the `CASE` statement serves a similar purpose, allowing for conditional logic based on specific values or conditions. It's the closest equivalent to a `SWITCH` statement in T-SQL.

In T-SQL, the `CASE` statement is primarily used within the context of queries, such as `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements. It is not used as a standalone control-of-flow structure like `IF` or `WHILE`.

In T-SQL, the `CASE` statement is specifically designed for conditional logic within the set-based operations of SQL queries. It's not a general-purpose control-of-flow statement like those found in procedural programming languages. Therefore, it cannot be used in the same way as an `if-else` or `switch` statement in languages like C# or Java, which control the flow of the program.

If you need to implement control-of-flow logic in T-SQL that is not directly tied to a query, you would typically use:

- IF...ELSE Statements: For conditional execution of T-SQL statements.
- WHILE Loops: For executing a set of statements repeatedly based on a condition.

**Understanding the CASE Statement as a SWITCH Equivalent**

The CASE statement can be used in two forms, which can mimic the behavior of a SWITCH statement **but only inside queries**:

1. Simple CASE (Equivalent to SWITCH): Compares an expression to a set of specific values.
2. Searched CASE: Evaluates a set of Boolean expressions.

Syntax of Simple CASE (SWITCH Equivalent)

```
CASE input_expression
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ...
    ELSE default_result
END
```

- `input_expression`: The expression to compare against the `WHEN` expressions.

Syntax of Searched CASE

```
CASE
    WHEN boolean_expression1 THEN result1
    WHEN boolean_expression2 THEN result2
    ...
    ELSE default_result
END
```

- Each `WHEN` clause contains a Boolean expression.

### Best Practices

- Avoid Complexity: Keep CASE statements simple for better readability.
- Performance Consideration: Be cautious with performance on large datasets.
- NULL Handling: CASE returns NULL if no conditions are met and there is no ELSE clause.
- Consistent Data Types: Ensure consistent data types in THEN and ELSE clauses.

### Summary

In T-SQL, the `CASE` statement functions as the closest equivalent to a `SWITCH` statement. It's a flexible tool for conditional logic, adaptable for various scenarios in SELECT, UPDATE, and ORDER BY clauses, enhancing SQL query functionality and dynamism.

## Simple CASE as SWITCH

**Simple CASE as SWITCH Example**:

```
SELECT
    EmployeeID,
    CASE DepartmentID
        WHEN 1 THEN 'Engineering'
        WHEN 2 THEN 'Human Resources'
        WHEN 3 THEN 'Sales'
        ELSE 'Other'
    END AS DepartmentName
FROM Employees;
```

- This mimics a SWITCH statement by assigning department names based on department IDs.

## Searched CASE (More Flexible)

ال searched case statement دي بتبحث عن قيمه معينه برضه سهله مجرد مسميات

**Searched CASE (More Flexible) Example:**

```
SELECT
    EmployeeID,
    CASE
        WHEN Salary <= 30000 THEN 'Entry Level'
        WHEN Salary BETWEEN 30001 AND 60000 THEN 'Mid Level'
        WHEN Salary > 60000 THEN 'Senior Level'
        ELSE 'Not Specified'
    END AS EmployeeLevel
FROM Employees;
```

- This uses searched CASE for more complex conditions, categorizing employees by salary.

## Using CASE in ORDER BY (Custom Sorting)

هنا بيرتب ال records عن طريق انه يقسمهم مجموعات كل مجموعه ليها رقم الرقم ده بيعبر عن الترتيب

زي مثلا انك عندك فواتير مبيعات ومشتريات ومرتجعات وعايز ترتبهم وتخلي المبيعات الأول فتديها الرقم واحد وبعدين المشتريات تديها رقم اتنين

فلما تعمل عليها order by هيرتبها حسب الرقم

الفكره انه بدل مايزود عمود في النتيجه بتاعت ال query لا هوا اخفي العمود بتاع الترتيب ده عن طريق انه يحط ال case statement بعد ال order by

في الاكسيل مثلا كانت بتقابلنا مشكلة اننا عايزين نرتب الداتا بترتيب معين

الترتيب ده مش ترتيب ابجدي ولا بيعتمد علي رقم فكنا بنزود عمود وكل نوع داتا معين نديه رقم وبعدين نرتب الجدول علي حسب العمود الجديد بتاعنا

دي نفس الفكره

**Using CASE in ORDER BY (Custom Sorting)**

```
SELECT Name, Salary
FROM Employees
ORDER BY
```

```
    CASE
        WHEN Salary > 50000 THEN 1
        ELSE 2
    END;
```

- Custom sorting of employees based on salary.

## CASE in UPDATE Statements (Conditional Data Modification)

شغل الكود ده

```
use C21_DB1;

CREATE TABLE Employees2 (
    Name VARCHAR(50),
    Department VARCHAR(50),
    Salary INT,
    PerformanceRating INT
);

INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 1',
'Marketing', 90651, 81);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 2',
'Marketing', 62945, 75);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 3', 'HR',
59665, 80);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 4', 'HR',
93279, 64);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 5', 'IT',
111884, 72);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 6', 'HR',
63199, 96);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 7',
'Marketing', 96372, 74);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 8', 'HR',
106907, 75);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 9', 'HR',
91444, 69);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 10', 'IT',
76801, 94);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 11',
'Marketing', 112522, 88);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 12', 'IT',
108024, 65);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 13', 'HR',
118334, 65);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 14',
'Marketing', 84143, 80);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 15', 'IT',
58163, 92);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 16', 'HR',
51840, 91);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 17',
'Marketing', 62225, 66);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 18',
'Marketing', 102274, 79);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 19',
'Marketing', 50149, 95);
INSERT INTO Employees2 (Name, Department, Salary, PerformanceRating) VALUES ('Employee 20',
'Marketing', 114694, 78);
```

الفكره كلها انه عايز يعدل عالمرتبات عن طريق الperformance

```
use C21_DB1;

UPDATE Employees2
SET Salary =
   CASE
      WHEN PerformanceRating > 90 THEN Salary * 1.15
      WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary *
1.10
         WHEN PerformanceRating BETWEEN 50 AND 74 THEN
Salary * 1.05
      ELSE Salary
   END;
```

**CASE in UPDATE Statements (Conditional Data Modification)**

```
UPDATE Employees2
SET Salary =
    CASE
        WHEN PerformanceRating > 90 THEN Salary * 1.15
        WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary * 1.10
                WHEN PerformanceRating BETWEEN 50 AND 74 THEN Salary * 1.05
        ELSE Salary
    END;
```

- Conditionally increasing salaries based on performance ratings.

## Nested Case Statements

Nested `CASE` statements in T-SQL (Transact-SQL) allow for complex conditional logic in SQL queries. Here's an example to illustrate how you can use nested `CASE` statements:

```
SELECT
    Name,
    Department,
    Salary,
    PerformanceRating,
    Bonus = CASE
                WHEN Department = 'Sales' THEN
                    CASE
```

```
                        WHEN PerformanceRating > 90 THEN Salary * 0.15
                        WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary * 0.10
                        ELSE Salary * 0.05
                    END
                WHEN Department = 'HR' THEN
                    CASE
                        WHEN PerformanceRating > 90 THEN Salary * 0.10
                        WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary * 0.08
                        ELSE Salary * 0.04
                    END
                ELSE
                    CASE
                        WHEN PerformanceRating > 90 THEN Salary * 0.08
                        WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary * 0.06
                        ELSE Salary * 0.03
                    END
            END
FROM Employees2;
```

In this example:

- The outer `CASE` statement evaluates the `Department` of each employee.
- Inside each `Department` case, there is a nested `CASE` statement that calculates the `Bonus` based on the `PerformanceRating`.
- Each department has different criteria for calculating the bonus.
- The `Bonus` is a percentage of the `Salary`, and the percentage varies based on the `PerformanceRating` and the `Department`.

## CASE statement within a GROUP BY

To demonstrate the use of a CASE statement within a `GROUP BY` clause , let's consider a scenario where we want to group employees by a custom category based on their `PerformanceRating`. For instance, we can categorize the performance as 'High', 'Medium', or 'Low'. Here's an example SQL query to achieve this:

```
SELECT
    PerformanceCategory,
    COUNT(*) AS NumberOfEmployees,
    AVG(Salary) AS AverageSalary
FROM
    (SELECT
        Name,
        Salary,
        CASE
            WHEN PerformanceRating >= 80 THEN 'High'
            WHEN PerformanceRating >= 60 THEN 'Medium'
            ELSE 'Low'
        END AS PerformanceCategory
    FROM Employees2) AS PerformanceTable
GROUP BY PerformanceCategory;
```

In this query:

- The inner `SELECT` statement creates a derived table (`PerformanceTable`) with an additional column `PerformanceCategory`. This column is determined by a `CASE` statement based on the `PerformanceRating`.
- The `CASE` statement categorizes performance into 'High' (rating >= 80), 'Medium' (rating >= 60), or 'Low' (else).
- The outer `SELECT` statement then groups the results by `PerformanceCategory`.
- It calculates the count of employees and the average salary in each performance category.

This query will provide a summary of how many employees fall into each performance category and what their average salary is.

**Quiz**

What is the purpose of the CASE statement in T-SQL?

- To serve as a control-of-flow structure
- To allow for conditional logic based on specific values
- To create loops in T-SQL
- To handle exceptions in T-SQL

In T-SQL, can the CASE statement be used as a standalone control-of-flow structure?

- Yes
- No, only used within queries

What is one best practice for using CASE statements in T-SQL?

- Avoid complexity for better readability
- Always use the searched case syntax
- Use case statements for control-of-flow logic
- Ensure inconsistent data types in then and else clauses

What is the result of a CASE statement if none of the conditions are met and there is no ELSE clause?

- An error is thrown
- NULL is returned
- The last WHEN condition is used
- The first WHEN condition is used

Can CASE statements be nested in T-SQL?

- No, T-SQL does not support nesting of CASE statements
- Yes, but only two levels deep
- Yes, there is no limit on the depth of nesting
- Yes, but only within stored procedures

What will happen if the expression in the WHEN clause of a CASE statement matches multiple conditions?

- All matching results are returned
- An error is thrown
- The result for the first matching condition is returned
- The result for the last matching condition is returned

Can a CASE statement be used in the ORDER BY clause of a SELECT statement in T-SQL?

- No, CASE statements are only for conditional logic in the SELECT clause
- Yes, to conditionally determine the order of rows
- Only if it is part of a stored procedure
- Yes, but only with a simple CASE statement

What is the result of a CASE statement if a searched condition uses a comparison with NULL using the '=' operator?

- It will always return true.
- It will always return false.
- It will not match, as NULL is not equal to any value, including itself.
- It causes a syntax error.

Which of the following operators can be used within a CASE statement's WHEN clause?

- LIKE, IN, BETWEEN
- Only = (equality)
- Only < and > (less than and greater than)
- All comparison and logical operators

## In T-SQL, can a CASE statement be used within an UPDATE statement?

- No, it's not allowed.
- Yes, but only to determine the column to be updated.
- Yes, to conditionally determine the value to set in a column.
- Yes, but it requires special permissions.

## Is it mandatory to have an ELSE clause in every CASE statement in T-SQL?

- Yes, it is required to complete the syntax.
- No, it is optional; if omitted, NULL is returned when no condition matches.
- No, but omitting it will cause a runtime error if no condition matches.
- Yes, but only in nested CASE statements.

## Can a CASE statement in T-SQL return different data types in different branches?

- Yes, but it is not recommended due to performance issues.
- No, all branches must return the same data type.
- Yes, but the SQL Server will implicitly convert them to a compatible type.
- No, it will result in a compilation error.

In T-SQL, which of the following is true about using a CASE statement in the GROUP BY clause?

- It's not allowed under any circumstances.
- It's allowed but will ignore NULL values.
- It's allowed but only with numeric data types.
- It's allowed and can be used to create groups based on conditional logic.

## WHILE Loops - Example 1 - Simple Counter

بالنسبه للloop فبيقولك انه ال t sql مافيهاش غير while loop واسمها while statement

وخليك فاكر انه ال begin وال end هما بديل الاقواس دي {}

```
declare @Counter as int =1

while @Counter<=5
begin
print('Count: '+cast(@Counter as varchar))
set @Counter =@Counter+1
end

set @Counter=5

while @Counter>=0
begin
print('Count: '+cast(@Counter as varchar))
set @Counter=@Counter-1
end
```

## WHILE Loops:

**Introduction to WHILE Loops in T-SQL**

A WHILE loop in T-SQL is a control-of-flow language construct that allows the execution of a specified block of SQL statements repeatedly as long as a specified condition is true.

**Basic Syntax**

```
WHILE [condition]
BEGIN
    -- SQL statements to be executed
END
```

- `condition`: A Boolean expression. If it evaluates to true, the loop continues; if false, the loop stops.

**Using a WHILE Loop**

WHILE loops are often used for repetitive tasks where the number of iterations isn't known beforehand or to iterate through records in a table one row at a time.

## There is No For loop or Do .. While loop In T-SQL

In T-SQL (Transact-SQL, used with Microsoft SQL Server), there are no `FOR` or `DO WHILE` statements as you would find in many other programming languages. The primary looping constructs available in T-SQL are the `WHILE` loop and the `CURSOR`, which is used to iterate over a result set row by row.

## Only WHILE Loop

The `WHILE` loop is the primary means for performing repeated actions in T-SQL and it works similarly to `WHILE` loops in other programming languages. It executes a block of statements as long as a specified condition is true.

## Example 2: Iterating Over a Table

عاوزين نلف عالموظفين في الجدول ونطبع اساميهم كلهم

هنعمل ده ازاي ؟

كالعاده عاوزين المتغيرات اللي هنخزن فيها الداتا

محتاجين اقل id في الجدول واعلي id في الجدول ومحتاجين متغير نخزن فيه اسم الموظف

```
declare @EmployeeID as INT
declare @EmployeeName as nvarchar(50)
declare @MaxID as int
```

```
select @EmployeeID = MIN(Employees.EmployeeID)
from Employees

select @MaxID = MAX(Employees.EmployeeID)
from Employees
```

بعد كده هنعمل ال while statement وفيها

هنحدد الشرط وده عادي

هنخزن اسم الموظف ونطبعه

هنخلي ال id يخزن رقم الموظف اللي بعده

```
WHILE @EmployeeID is not null and @EmployeeID <= @MaxID
begin
select @EmployeeName=Employees.Name
from Employees
where Employees.EmployeeID =@EmployeeID

print(@EmployeeName)

select @EmployeeID = MIN(Employees.EmployeeID)
from Employees
where Employees.EmployeeID > @EmployeeID

end
```

```
declare @EmployeeID as INT
declare @EmployeeName as nvarchar(50)
declare @MaxID as int

select @EmployeeID = MIN(Employees.EmployeeID)
from Employees

select @MaxID = MAX(Employees.EmployeeID)
from Employees

WHILE @EmployeeID is not null and @EmployeeID <= @MaxID
begin
select @EmployeeName=Employees.Name
from Employees
where Employees.EmployeeID =@EmployeeID

print(@EmployeeName)

select @EmployeeID = MIN(Employees.EmployeeID)
from Employees
where Employees.EmployeeID > @EmployeeID

end
```

## 03 Example 3 - Loop with Conditional Exit

عاوزين نعمل متغير يكون فيه الرصيد ونعمل while loop بحيث اننا كل مرة نسحب 100 ونعرض الرصيد بعد السحب ولو مافيش فلوس تكفي يطلعلي رساله ويطبعلي الرصيد الباقي كام

```
declare @Balance as decimal(10,2) =950
declare @Withdrowal as decimal(10,2)=100

while @Balance>0
begin

if @Balance >= @Withdrowal
begin
set @Balance=@Balance-@Withdrowal
print'your current Balance is: '+cast(@Balance as varchar)
end

else
begin
print'in suffitent fund'
print'your current Balance is: '+cast(@Balance as varchar)
break
end

end
```

## Example 4 - Nested While Loops - 10 x 10 Multiplication Table

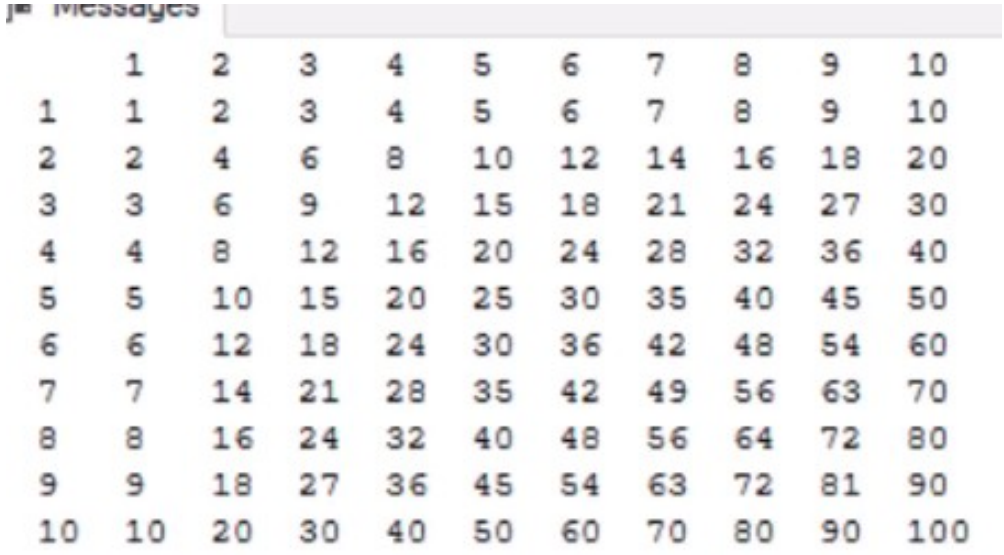
عباره عن loop جوه loop ومثال عليها جدول الضرب

```
declare @row as int=1
declare @col as int


while @row<=10
begin

if @row <= 10
BEGIN
SET @col=1
END

WHILE @col <=10
BEGIN
PRINT CAST(@row as varchar) + ' '+' * '+  CAST(@col as
varchar) + ' = '+cast((@row*@col) as varchar)
set @col=@col+1
END
set @row=@row+1
```

```
end
```

## Example 5 - 10 x 10 Matrix Multiplication Table

عاوزين نطبع جدول الضرب بالطريقه دي



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
| 3 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 |
| 4 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 |
| 5 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
| 6 | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 |
| 7 | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 |
| 8 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 |
| 9 | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 |
| 10 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

دي طريقه حله

```
-- 10 x 10 Matrix Multiplication Table

DECLARE @row INT = 1;
DECLARE @col INT;
DECLARE @result INT;
DECLARE @rowString VARCHAR(255);
DECLARE @headerString VARCHAR(255);

-- Create the header row for the columns
SET @headerString = CHAR(9); -- Starting with a tab for the row header space
SET @col = 1;
WHILE @col <= 10
BEGIN
    SET @headerString = @headerString + CAST(@col AS VARCHAR) + CHAR(9); -- Append column headers
    SET @col = @col + 1;
END
PRINT @headerString;

-- Generate the multiplication table
WHILE @row <= 10
BEGIN
    SET @col = 1;
    SET @rowString = CAST(@row AS VARCHAR) + CHAR(9); -- Start each row with the row number

    WHILE @col <= 10
    BEGIN
        SET @result = @row * @col;
        SET @rowString = @rowString + CAST(@result AS VARCHAR) + CHAR(9); -- Append multiplication
results
        SET @col = @col + 1;
    END

    PRINT @rowString; -- Print the row
    SET @row = @row + 1;
END
```

ودي طريقة حلي

```
declare @row as int=1
declare @col as int
declare @RowValue as nvarchar(255)=char(9)

while @row<=10
begin
set @RowValue=@RowValue+ cast(@row as varchar)+char(9)
set @row=@row+1
end

print(@RowValue)

set @RowValue=null
set @row=1


while @row<=10
begin

if @row<=10
begin
set @col=1
set @RowValue=cast (@row as varchar)+char(9)
end

while @col<=10
begin
set @RowValue=@RowValue+cast ((@row*@col) as varchar)+char(9)
set @col=@col+1
end
print ( @RowValue)
set @row=@row+1
set @RowValue=cast (@row as varchar)+char(9)
end
```

## Example 6 - BREAK and CONTINUE Statements

ال break وال continue زيهم زي أي لغة برمجه تانيه ال break بيطلع من ال loop كلها وال continue بتدخل عالدورة اللي بعدها

BREAK and CONTINUE statements in T-SQL:

`BREAK` and `CONTINUE` statements in Transact-SQL (T-SQL), which is used with Microsoft SQL Server. These statements are primarily used within loops to control the flow of execution.

## Understanding `BREAK` and `CONTINUE` in T-SQL

1. Introduction to Loops in T-SQL

- Loops in T-SQL, like in other programming languages, are used to execute a set of statements repeatedly until a specified condition is met.
- The most common loop in T-SQL is the `WHILE` loop.

2. `BREAK` Statement

- Purpose: The `BREAK` statement is used to immediately exit the loop, regardless of whether the loop condition is still true.
- Usage: Typically used when a certain condition is met inside the loop, and there is no need to continue looping.

**Example:**

```
DECLARE @counter INT = 1;

--Break Example
--This loop will print numbers from 1 to 5.
-- When the counter reaches 5, the BREAK statement exits the loop.
 PRINT 'Break Example: ' ;
WHILE @counter <= 10
BEGIN
    PRINT 'Counter: ' + CAST(@counter AS VARCHAR);
    IF @counter = 5
    BEGIN
        BREAK; -- Exits the loop when counter reaches 5
    END
    SET @counter = @counter + 1;
END
```

3. `CONTINUE` Statement

- Purpose: The `CONTINUE` statement is used to skip the rest of the loop body and immediately start the next iteration of the loop.
- Usage: Commonly used to skip certain iterations based on a condition.

**Example:**

```sql
--This loop will print only odd numbers between 1 and 10.
-- When the counter is even, the CONTINUE statement skips to the next iteration.
DECLARE @counter INT = 1;
PRINT 'Continue Example: ' ;
set @counter=1;

WHILE @counter <= 10
BEGIN
    SET @counter = @counter + 1;
    IF @counter % 2 = 0
    BEGIN
        CONTINUE; -- Skips the current iteration for even numbers
    END
    PRINT 'Counter: ' + CAST(@counter AS VARCHAR);
END
```

4. Key Differences and Usage Scenarios

- Use `BREAK` when you want to exit the loop entirely.
- Use `CONTINUE` when you want to skip the current iteration and proceed with the next iteration.

## Important: There is No For Loop or Do While Statement In T-SQL

وهنا بيقولك انه ماعندناش في ال tsql غير ال while loop ومفيش حاجه تانيه زي ال for loop او ال do while loop

## There is No For loop or Do .. While loop In T-SQL

In T-SQL (Transact-SQL, used with Microsoft SQL Server), there are no `FOR` or `DO WHILE` statements as you would find in many other programming languages. The primary looping constructs available in T-SQL are the `WHILE` loop and the `CURSOR`, which is used to iterate over a result set row by row.

## Only WHILE Loop

The `WHILE` loop is the primary means for performing repeated actions in T-SQL and it works similarly to `WHILE` loops in other programming languages. It executes a block of statements as long as a specified condition is true.

## Quiz

## What is the purpose of a WHILE loop in T-SQL?

- To execute a block of SQL statements repeatedly
- To stop the execution of a block of SQL statements
- To define a Boolean expression

## What is the syntax of a WHILE loop in T-SQL?

- WHILE [condition] END
- WHILE condition BEGIN END
- WHILE [condition] BEGIN
- WHILE condition END

## Is there a FOR loop in T-SQL?

- Yes
- No

## What is the primary looping construct in T-SQL?

- FOR loop
- DO WHILE loop
- WHILE loop
- CURSOR

**BEGIN...END Blocks in T-SQL**

هنا مفيش جديد بيقولك انه ال begin وال end زي الاقواس دي {} في لغات البرمجه التانيه

# BEGIN...END Blocks in T-SQL

**Introduction to BEGIN...END Blocks in T-SQL**

The `BEGIN...END` block in T-SQL is a control-of-flow language construct used to group a series of T-SQL statements into a single block. This is particularly useful for defining the body of control-of-flow statements like `IF...ELSE`, `WHILE`, and others.

`BEGIN...END` blocks are similar to { } in other programming languages.

**Syntax**

The basic syntax of a `BEGIN...END` block is:

```
BEGIN
    -- Series of T-SQL statements
END
```

Usage of BEGIN...END Blocks

1. In Control-of-Flow Constructs: To define the scope of statements within control structures like `IF`, `ELSE`, and `WHILE`.

```
IF @condition = TRUE
BEGIN
    -- Statements to execute if condition is true
END
```

1. To Group Statements: Group multiple T-SQL statements so that they are executed together as a unit.
2. In Stored Procedures and Triggers: To define the set of statements that make up a stored procedure or trigger.

**Characteristics**

- Scope Definition: `BEGIN...END` blocks define the start and end of a statement group.
- Nested Blocks: These blocks can be nested inside one another.

**Examples**

- Simple BEGIN...END Block:

```
BEGIN
    PRINT 'Hello, World!';
    SELECT * FROM Employees;
END
```

- Nested Blocks:

```
IF @x > 10
BEGIN
    PRINT 'X is greater than 10';
    IF @y < 20
    BEGIN
        PRINT 'Y is less than 20';
    END
END
```

- In WHILE Loops:

```
WHILE @counter < 10
BEGIN
    PRINT @counter;
    SET @counter = @counter + 1;
END
```

## Error Handling in T-SQL - Try .. Catch

القوسين بنعوض عنهم بإيه؟

ايوه كده بال begin وال end

في ال try وال catch بتكتب try بعد ال begin وال end بتوع اول block بعد ال begin وال end بتوع تاني block

تقدر تستخدمها في انك تبعت رسالة الخطأ للبرنامج بتاعك بدل ماتقعد تلف حوالين نفسك

اعمل الجدول ده عندك الأول

```
use C21_DB1
```

```
create table Employees3(
EmployeeID int Primary key,
Name nvarchar(100),
Position nvarchar(100)
)
```

وده مثال عال TRY وال CATCH

```
begin try
Insert into Employees3(EmployeeID,Name,Position)
values (1,'John Doe','Sales Manager')

Insert into Employees3(EmployeeID,Name,Position)
values (1,'Jane Smith','Marketing Manager')

end try
begin catch
print 'An Error Occured: '+ ERROR_MESSAGE()
end catch
```

## Error Handling in T-SQL:

**Introduction**

Error handling in T-SQL is a crucial aspect of writing robust SQL code. It allows you to gracefully handle unexpected events and errors that occur during the execution of SQL scripts.

Why is Error Handling Important?

- Prevents Data Corruption: Proper error handling can prevent partial updates and maintain data integrity.
- User-Friendly Feedback: It can provide meaningful information to the user or calling application about what went wrong.
- Flow Control: It allows the code to continue running or to stop based on the severity of the error.

**TRY...CATCH in T-SQL**

The primary mechanism for error handling in T-SQL is the `TRY...CATCH` construct.

- TRY Block: You place the T-SQL code that might cause an error inside a `TRY` block. If an error occurs, execution is passed to the associated `CATCH` block.
- CATCH Block: The `CATCH` block contains code that runs if an error occurs in the `TRY` block. It can log the error, roll back transactions, and take other appropriate actions.

### Syntax

```
BEGIN TRY
    -- T-SQL statements that may cause an error
END TRY
BEGIN CATCH
    -- Error handling code
END CATCH
```

### Error Functions

Within the `CATCH` block, you can use functions to get detailed error information:

- `ERROR_NUMBER()`: Returns the error number.
- `ERROR_SEVERITY()`: Returns the severity.
- `ERROR_STATE()`: Returns the error state number.
- `ERROR_PROCEDURE()`: Returns the name of the stored procedure or trigger where the error occurred.
- `ERROR_LINE()`: Returns the line number where the error occurred.
- `ERROR_MESSAGE()`: Returns the complete text of the error message.

### Example: Using TRY...CATCH

Let's consider a scenario where you are inserting data into a table and want to handle potential errors.

```
-- Assume we have a table called 'Employees' with a unique constraint on 'EmployeeID'
CREATE TABLE Employees3 (
    EmployeeID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Position NVARCHAR(100)
```

```
);



BEGIN TRY
    -- Insert a record into the Employees table
    INSERT INTO Employees3 (EmployeeID, Name, Position) VALUES (1, 'John Doe', 'Sales Manager');


    -- Attempt to insert a duplicate record which will cause an error
    INSERT INTO Employees3 (EmployeeID, Name, Position) VALUES (1, 'Jane Smith', 'Marketing Manager');
END TRY
BEGIN CATCH
    -- Handle the error
    PRINT 'An error occurred: ' + ERROR_MESSAGE();
    -- Rollback the transaction if any
END CATCH
```

In this example, the second `INSERT` statement will fail because it violates the unique constraint on `EmployeeID`. The error is caught in the `CATCH` block where a message is printed, and you could also add logic to roll back a transaction if necessary.

### Best Practices

- Use TRY...CATCH for all your transactions: Protect your data integrity by wrapping transactions in a `TRY...CATCH` block.
- Log Errors: Always log errors for later analysis, which can help in understanding what went wrong.
- Provide User Feedback: Where appropriate, pass back information to the user, but avoid revealing sensitive information about the database structure or system.

### Conclusion

Effective error handling in T-SQL is essential for creating reliable, robust applications. Using `TRY...CATCH` blocks allows you to handle errors gracefully and ensure that your T-SQL scripts execute as intended, even when faced with the unexpected.

**Quiz 1**

What is the primary mechanism for error handling in T-SQL?

- TRY. . . CATCH
- BEGIN TRANSACTION
- IF. . . ELSE
- SELECT. . . INTO

What does the CATCH block in a TRY. . . CATCH construct contain?

- Code that runs if an error occurs
- Code that runs before the error occurs
- Code that always executes
- Code that is skipped upon error

Which function returns the complete text of the error message?

- ERROR_NUMBER()
- ERROR_SEVERITY()
- ERROR_STATE()
- ERROR_MESSAGE()

## Error Functions

فيه functions مخصوصة لل errors منها

ERROR_NUMBER رقم الخطأ

ERROR_SEVERITY خطورة الخطأ

ERROR_STATE ال ERROR_NUMBER بيتميز بحاجه اسمها error state

ERROR_PROCEDURE بترجعلك اسم ال FUNCTION اللي حصل فيها الخطأ

ERROR_LINE بترجعلك السطر اللي حصل فيه الخطأ

ERROR_MESSAGE بترجعلك الdescribption بتاع الخطأ

```
BEGIN TRY
    -- Intentional division by zero error
    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
```

## Understanding Error Functions in T-SQL

**Introduction**

In T-SQL (Transact-SQL used in Microsoft SQL Server), error handling is an essential aspect of writing robust and reliable database applications. SQL Server provides several functions that can be used within a CATCH block of a TRY...CATCH construct to retrieve detailed information about errors. Understanding these functions is crucial for diagnosing and responding to errors effectively.

**Error Functions Overview**

- **ERROR_NUMBER()**
    - Purpose: Returns the error number of the error that caused the CATCH block to be executed.
    - Usage: Useful for identifying the specific error that occurred.
- **ERROR_SEVERITY()**
    - Purpose: Returns the severity level of the error.

    - Usage: Helps in understanding the nature and seriousness of the error. Severity levels range from 0 to 25.

- **ERROR_STATE()**
    - Purpose: Returns the state number of the error.
    - Usage: Useful for providing additional information about the error or to distinguish between errors with the same number.
- **ERROR_PROCEDURE()**
    - Purpose: Returns the name of the stored procedure or trigger in which the error occurred.
    - Usage: Essential for identifying the source of the error in complex systems with multiple procedures and triggers.
- **ERROR_LINE()**
    - Purpose: Returns the line number where the error occurred.
    - Usage: Helps in pinpointing the exact location in the code where the error was raised, facilitating quicker debugging.
- **ERROR_MESSAGE()**
    - Purpose: Provides the complete text of the error message.
    - Usage: Offers a detailed description of the error, which is valuable for understanding what went wrong.

### Practical Example

To illustrate the use of these error functions, let's consider a simple code that can generate an error:

```
BEGIN TRY
    -- Intentional division by zero error
    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
```

When this code is executed, it will raise a division by zero error. The CATCH block will catch this error and use the error functions to return detailed information about the error.

**Conclusion**

Understanding and using these error functions effectively allows T-SQL developers to write more reliable and maintainable code by providing comprehensive error diagnostics. This information can be used for logging, debugging, or even to inform users about the nature of an issue in a more user-friendly manner. Remember, thorough error handling is a hallmark of high-quality database programming.

## Quiz 2

What is the purpose of the ERROR_NUMBER() function?

- Returns the line number where the error occurred
- Returns the severity level of the error
- Returns the error number of the error that caused the CATCH block to be executed
- Provides the complete text of the error message

## What is the purpose of the ERROR_SEVERITY() function?

- Returns the error number of the error that caused the CATCH block to be executed
- Returns the line number where the error occurred
- Returns the severity level of the error
- Provides the complete text of the error message

## What is the purpose of the ERROR_STATE() function?

- Returns the severity level of the error
- Returns the line number where the error occurred
- Returns the state number of the error
- Provides the complete text of the error message

## What is the purpose of the ERROR_PROCEDURE() function?

- Returns the error number of the error that caused the CATCH block to be executed
- Returns the line number where the error occurred
- Returns the state number of the error
- Returns the name of the stored procedure or trigger in which the error occurred

What is the purpose of the ERROR_LINE() function?

- Returns the severity level of the error
- Returns the name of the stored procedure or trigger in which the error occurred
- Returns the line number where the error occurred
- Provides the complete text of the error message

What is the purpose of the ERROR_MESSAGE() function?

- Returns the state number of the error
- Returns the line number where the error occurred
- Provides the complete text of the error message
- Returns the name of the stored procedure or trigger in which the error occurred

How can error functions be beneficial for T-SQL developers?

- They help in understanding the nature and seriousness of the error
- They can be used to pinpoint the exact location in the code where the error was raised
- They provide comprehensive error diagnostics for logging, debugging, and informing users
- All of the above

## THROW Statement

ال throw statement وفكرته موجوده في أي لغة برمجه وفكرته انك بتعمل ال error بنفسك عشان يشيك علي حاجه معينه

طريقته انك تكتب كلمة thow[] وبين القوسين بتكتب رقم الخطأ والرسالة وال state

رقم الخطأ بتحطه من عندك وتقدر تحط رقم من 50 الف لحد 2 مليار وشوية

الstate دي اعتبرها مجموعه فرعيه من رقم الخطأ يعني ممكن تعمل عدة أخطاء بنفس الرقم عشان تصنفهم كمجموعه مع بعض وبعدين تمسك كل خطأ وتديله رقم مميز ليه وده تقدر تديله من صفر لحد 255

لو ماكتبتش حاجه مكان ال error number وال message وال state وجيت شغلت الكود هيديلك ال error الأصلي

ال throw statement بتستخدمها داخل ال catch statement لو حصل خطأ بيطلع من ال catch

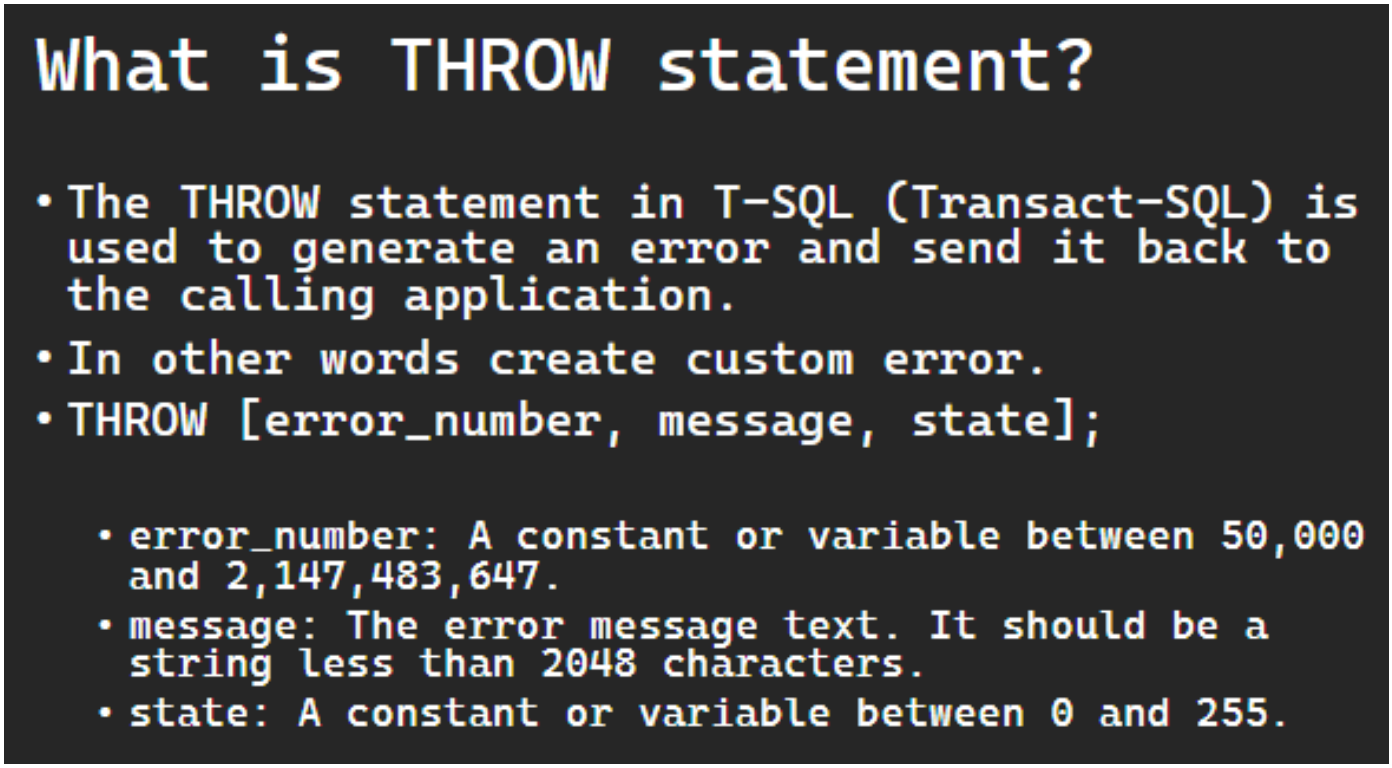
ولو عندك nested catch statements وجيت عمل thorw statement في ال catch اللي جوه هتوديك لل catch الخارجيه ولو عملت thorw في الخارجيه هتطلعك منها

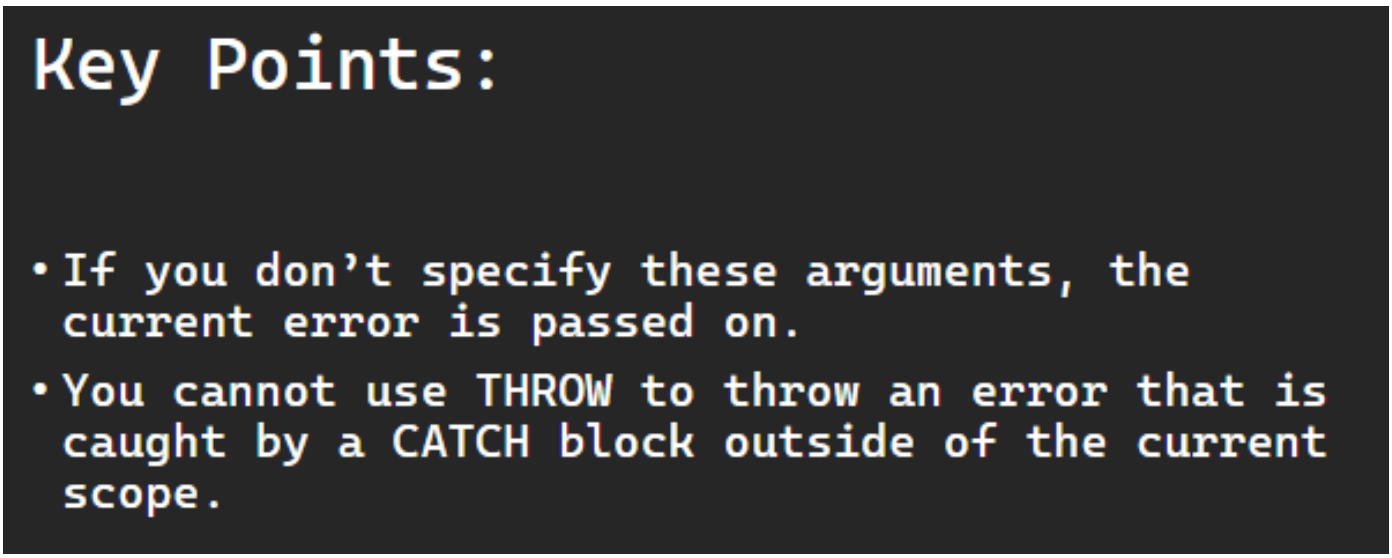
تعالي اعمل الجدول ده

```
use C21_DB1;

CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    StockQuantity INT
);

Go

INSERT INTO Products (ProductID, StockQuantity) VALUES (1, 100);
INSERT INTO Products (ProductID, StockQuantity) VALUES (2, 50);
INSERT INTO Products (ProductID, StockQuantity) VALUES (3, 75);
```

وده المثال

```
       use C21_DB1;

    declare @NewStockQty INT;

       set @NewStockQty=-5;


    -- Start a TRY block
    BEGIN TRY
        -- Check if NewStockQty is negative
        IF @NewStockQty < 0
            THROW 51000, 'Stock quantity cannot be negative.', 1;

        -- Proceed with updating stock (example code)
        UPDATE Products SET StockQuantity = @NewStockQty WHERE ProductID = 1;
    END TRY

    -- Start a CATCH block to handle the error
    BEGIN CATCH
        SELECT
            ERROR_NUMBER() AS ErrorNumber,
            ERROR_MESSAGE() AS ErrorMessage;
    END CATCH
```




The `THROW` statement in T-SQL (Transact-SQL) is used to generate an error and send it back to the calling application. It's particularly useful for handling errors in stored procedures, triggers, or batches. Here's a basic lesson on how to use `THROW`, along with an example.

## Understanding `THROW` Statement

1. Purpose: Used to raise an exception and transfer control to a CATCH block of a TRY...CATCH construct in your SQL code.
2. Syntax:

```
THROW [error_number, message, state];
```

- error_number: A constant or variable between 50000 and 2147483647.
- message: The error message text. It should be a string less than 2048 characters.
- state: A constant or variable between 0 and 255.

**Key Points:**

- 
  - If you don’t specify these arguments, the current error is passed on.
  - You cannot use `THROW` to throw an error that is caught by a CATCH block outside of the current scope: This means that if an error is raised using `THROW` within a `TRY` block, it must be caught by the corresponding `CATCH` block that is in the same scope as the `TRY` block. If there's a nested `TRY...CATCH` (one inside another), an error thrown inside the inner `TRY` block cannot be caught by the `CATCH` block of the outer `TRY...CATCH` structure. It has to be caught within the same level of nesting.

## Example: Updating Product Inventory

**Scenario:**

We have a this code that updates the stock quantity of a product in the inventory. The procedure should raise an error if the new stock quantity is negative, as this would not be a valid scenario in most inventory systems.

**Stored Procedure with `THROW`:**

```
declare @NewStockQty INT;



      set @NewStockQty=-5;


  -- Start a TRY block
  BEGIN TRY
      -- Check if NewStockQty is negative
      IF @NewStockQty < 0
          THROW 51000, 'Stock quantity cannot be negative.', 1;
```

```
    -- Proceed with updating stock (example code)
    UPDATE Products SET StockQuantity = @NewStockQty WHERE ProductID = 1;
END TRY



-- Start a CATCH block to handle the error
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

Explanation:

- In this code we first enter a TRY block.
- We then check if `@NewStockQty` is negative. If it is, we use the `THROW` statement to raise an error. The error number is 51000, and we provide a custom error message stating that the stock quantity cannot be negative. The state is set to 1.
- If the stock quantity is valid, the procedure updates the `Products` table with the new stock quantity.
- If an error is thrown, control passes to the CATCH block, which captures and returns the error information.

This example demonstrates how `THROW` can be effectively used to ensure data integrity and prevent invalid operations in database procedures. It's particularly useful in scenarios where business rules dictate specific constraints that are not directly enforced by the database schema itself.

Remember, proper error handling in SQL is crucial for writing robust and reliable applications. The `THROW` statement is a powerful tool for this purpose, especially in combination with the TRY...CATCH construct.

**Quiz 3**

## What is the purpose of the THROW statement in T-SQL?

- To generate an error and send it back to the calling application
- To handle errors in stored procedures, triggers, or batches
- To update the stock quantity of a product in the inventory
- To capture and return error information in a CATCH block

## What is the syntax of the THROW statement?

- THROW [error_number, message, state]
- THROW (error_number, message, state)
- THROW error_number, message, state
- THROW { error_number: error_number, message: message, state: state }

## What is the purpose of the error_number argument in the THROW statement?

- To specify a constant or variable between 50000 and 2147483647
- To specify the error message text
- To specify a constant or variable between 0 and 255
- To specify the state of the error

How many characters can the error message text be in the THROW statement?

- Less than 2048 characters
- Exactly 2048 characters
- Any number of characters
- Less than 1024 characters

Can the THROW statement be used to throw an error that is caught by a CATCH block outside of the current scope?

- No, it can only throw an error that is caught by a CATCH block within the current scope
- Yes, it can throw an error that is caught by a CATCH block outside of the current scope
- It depends on the error number and message specified
- Throwing an error to a CATCH block outside of the current scope is not supported

## @@ERROR Function

بيقولك انه في الإصدارات القديمة بتاعت ال sql server مكانش فيه try catch بس كان فيه طريقه بيعرفوا بيها ان كان فيه خطأ حصل ولا لا

فكانوا بيستخدموا حاجه اسمها @@ERROR ودي بترجعلك رقم الخطأ اللي حصل نتيجة اخر امر statement نفذتها

فكنت بعد كل statement كنت بتشيك لو ال @@ERROR قيمة لاتسوي صفر معناه انه فيه خطأ حصل بس مش بيمنع ال crash

هنا بيعلمهالك عشان لو عندك عميل بيستخدم اصدار قديم تبقي عارفها

```
use C21_DB1;

-- Attempt to insert an invalid record into a table
INSERT INTO Departments (DepartmentID, Name)
VALUES (1, 'Business'); -- Assume 'DepartmentID' is a
```

```
primary key and '1' already exists

DECLARE @ErrorNumber INT = @@ERROR;
-- Check if the previous statement caused an error
IF @ErrorNumber <> 0
BEGIN
   -- Handle the error
   PRINT 'An error occurred during the insert operation.';
   -- You can also capture the specific error number and
store it or use it in logic
   PRINT 'The error number is: ' + CAST(@ErrorNumber AS
VARCHAR);
END
```

The `@@ERROR` function in T-SQL (Transact-SQL used in Microsoft SQL Server) is a system function that returns the error number of the last Transact-SQL statement executed. It's an older method of error checking in SQL Server, often used before the introduction of the TRY...CATCH construct.

## Understanding `@@ERROR`

1. Purpose: `@@ERROR` provides the error number of the last T-SQL statement that was executed. If the last statement was successful, `@@ERROR` returns 0.
2. Usage:
    - 
    - It must be checked immediately after the statement that might cause an error, because any subsequent statement will reset `@@ERROR` to 0 if it executes successfully.
    - `@@ERROR` is often used in older scripts or in systems where TRY...CATCH is not available or applicable.

Syntax:

```
SELECT @@ERROR
```

## Limitations and Modern Alternative

While `@@ERROR` is simple and straightforward, it has significant limitations compared to the modern TRY...CATCH construct:

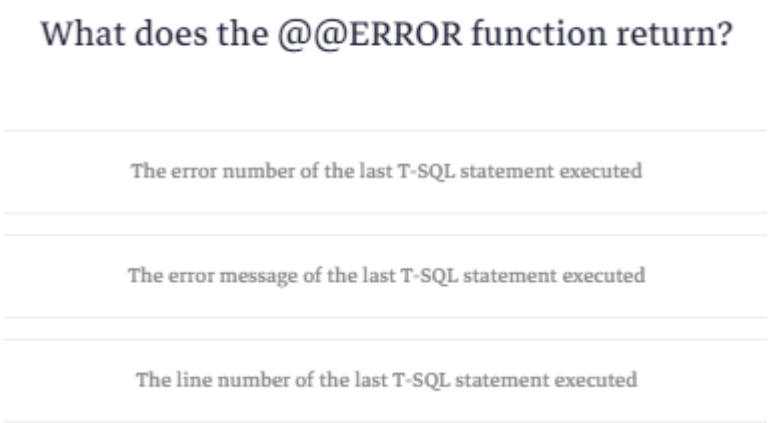
- It only captures the error number of the last statement executed, so it must be checked immediately after the relevant SQL statement.
- It doesn't provide detailed error information like the error message, line number, or severity.
- Managing `@@ERROR` checks after every statement can make the code cluttered and harder to maintain.

The modern approach is to use TRY...CATCH blocks, which provide a more structured and comprehensive way of handling errors in SQL Server.

## Conclusion

`@@ERROR` is useful for backward compatibility and in simple scripts where detailed error information is not required. However, for new developments, it's generally recommended to use TRY...CATCH blocks for more robust error handling.

**Quiz 4**



When should @@ERROR be checked?

- Immediately after the statement that might cause an error
- Before executing any SQL statement
- After executing all SQL statements

What does it mean if @@ERROR returns a value different from 0?

- The operation was successful
- An error occurred during the operation
- The statement is still being executed

What are the limitations of using @@ERROR?

- It provides detailed error information
- It captures the error message and line number
- It only captures the error number of the last statement executed

What is the modern alternative to using @@ERROR?

- TRY. . . CATCH blocks
- IF statements
- WHILE loops

## @@ROWCOUNT

@@ROWCOUNT بيرجعلك عدد الصفوف اللي اتاثرت بالQUERY بتاعتك

```
UPDATE Employees set DepartmentID=3 where DepartmentID=4

select @@ROWCOUNT as RowsAffected
```

`@@ROWCOUNT` is a system function in T-SQL (Transact-SQL used in Microsoft SQL Server) that returns the number of rows affected by the last statement executed. This function is commonly used to determine how many rows were impacted by the previous operation, such as an INSERT, UPDATE, DELETE, or SELECT statement.

## Understanding `@@ROWCOUNT`

1. Purpose: To get the number of rows affected by the most recently executed statement in your SQL script or batch.
2. Usage:
    - 
    - It must be checked immediately after the statement whose impact you want to measure, because any subsequent statement, including something as simple as a `PRINT` statement, will reset `@@ROWCOUNT` to the number of rows affected by that subsequent statement.
    - `@@ROWCOUNT` is often used to verify the success of a statement or to take conditional action depending on the number of rows affected.

1. Syntax:

```
SELECT @@ROWCOUNT
```

## Example Usage

Consider a scenario where you update records in a table and want to check how many rows were updated.

```
UPDATE Employees SET DepartmentID = 3 WHERE DepartmentID =4;

SELECT @@ROWCOUNT AS RowsAffected;
```

In this example:

- Immediately after, `@@ROWCOUNT` is used to return the number of rows that were updated by the `UPDATE` statement.

## Practical Considerations

- Immediate Check: Always check `@@ROWCOUNT` immediately after the relevant SQL statement, as its value is reset after each statement.
- Use in Conditional Logic: It's often used in conditional logic, such as in IF statements, to take different actions depending on the number of rows affected by a previous operation.
- Zero Rows Affected: If no rows are affected by the previous operation, `@@ROWCOUNT` returns 0. This can be useful to check whether a conditional update or delete actually changed any data.
- Compatibility: `@@ROWCOUNT` is widely supported and is a standard part of T-SQL, making it compatible with various versions of SQL Server.

## Conclusion

`@@ROWCOUNT` is a valuable tool in T-SQL for understanding the impact of SQL statements and controlling the flow of scripts based on how many rows are affected by certain operations. It's especially useful in data manipulation scenarios and in ensuring the effectiveness of SQL commands.

**Quiz**

What does @@ROWCOUNT return in T-SQL?

The number of rows affected by the last statement executed.

The total number of rows in a table.

The number of columns in a table.

The average number of rows affected by all statements executed.

## When should you check the value of @@ROWCOUNT?

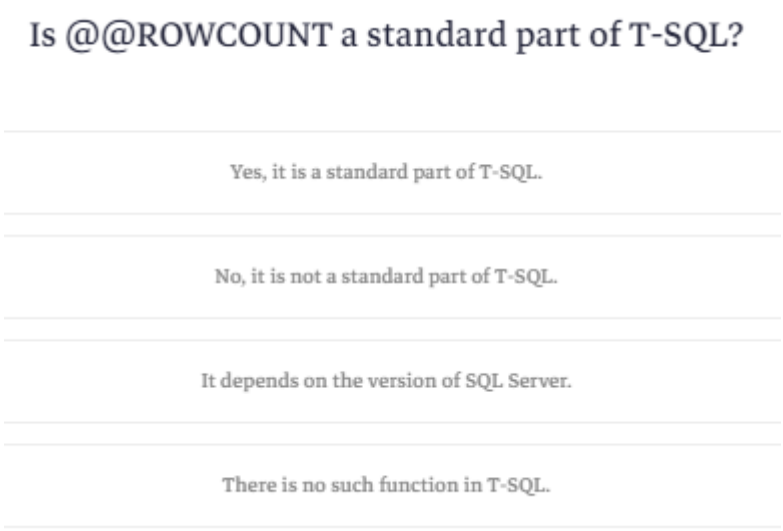
- Immediately after the relevant SQL statement.
- Before executing any SQL statement.
- At the end of the SQL script or batch.
- It doesn't matter when you check the value of @@ROWCOUNT.

## What can @@ROWCOUNT be used for?

- Verifying the success of a statement.
- Taking conditional action based on the number of rows affected.
- Calculating the average number of rows affected by all statements.
- Determining the total number of rows in a table.

## What does @@ROWCOUNT return if no rows are affected?

- The total number of rows in the table.
- The number of columns in the table.
- 0
- It returns an error message.




## Introduction to Transactions

لو انت دلوقتي عندك داتا بيز خاصه ببنك معينه ومحتاج تحول فلوس من عميل لعميل تاني

فا انت هنا محتاج تعمل 2 update statements عشان تخصم من حساب عميل وتضيف للعميل التاني ومحتاج تسجل record جديد بانه فيه عملية تحويل حصلت من حساب فلان لحساب علان والقيمه كذا والتاريخ والوقت كذا

هنا ماينفعش تعمل كل عملية لوحدها لازم الثلاثه يتنفذوا مع بعض ولو query واحده حصل فيها مشكله ماينفذش الباقي عشان مايبوظش الدنيا

عشان كده عملولك حاجه اسمها transactions وهيا فكرتها انها بتحتفظ بالداتا قبل التعديل بحيث لو حصل خطأ في query معينه يعمل rollback ويرجع الداتا زي ماكانت ولو كانت كل العمليات ناجحه بيعمل commit

وبيقولك انه ال transactions في ال sql لازم كل الأوامر اللي في ال transaction يتحقق فيها كلها حاجه اسمها ACID properties ياما مايتحققش في ولا واحده من الأوامر

طيب ايه هيا ال ACID properties دي؟

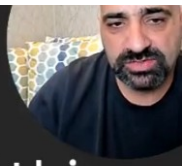
قالك هيا اختصار لعدة مفاهيم هما:-

1- Atomicity :- وهيا انه كل الأوامر تتنفذ كلها او ماتتنفذش خالص
2- consistency :- انك تحترم قوعد الداتابيز زي ال referential integrity وغيرها
3- Isolation :- كل عملية منفصله عن التانيه اه لكن كلهم بيتنفذوا في نطاق ال transaction عشان مفيش عمليه تتداخل مع عمليه تانيه وعشان لو حصل خطأ في واحده منهم نقدر نعمل rollback
4- Durability :- هنا هو بيخزن الداتا في log file بحيث انك لو نفذت الأوامر كلها وجاي تعمل comitt والكهرباء اتقطعت مثلا هتيجي تشغل الداتا بيز تاني هتروح تاخد الداتا من ال log file وتعملupdate للداتابيز

# What is transaction?

- A transaction in SQL is a series of database operations that are treated as a single logical unit.
- It ensures that either all operations within it are executed or none are.
- ACID Properties: Transactions in SQL adhere to ACID properties - Atomicity, Consistency, Isolation, Durability.

# ACID



- Atomicity: This ensures that all operations within a transaction are treated as a single unit. Either all of them are executed successfully, or none are. If any part of the transaction fails, the entire transaction is rolled back (undone), maintaining data integrity.

- Consistency: Consistency ensures that a transaction brings the database from one valid state to another. Integrity constraints are maintained so that the database remains consistent before and after the transaction.

- Isolation: Isolation ensures that transactions are securely and independently processed at the same time without interference, but the results of the transaction are as if the transactions were processed sequentially. This prevents transactions from reading intermediate (and possibly inconsistent) data.

- Durability: Durability guarantees that once a transaction has been committed, it will remain so, even in the event of a system failure. This means that the changes made by the transaction are permanently stored in the database.

# Transactions in T-SQL:

**Introduction to Transactions**

- Definition: A transaction in SQL is a series of database operations that are treated as a single logical unit. It ensures that either all operations within it are executed or none are.
- ACID Properties: Transactions in SQL adhere to ACID properties - Atomicity, Consistency, Isolation, Durability.

**Why Use Transactions?**

- Data Integrity: Critical for operations that must not be partially completed, such as bank transfers.
- Error Handling: Transactions help in managing errors and maintaining database consistency.

**ACID** is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. It's a set of properties that guarantee that database transactions are processed reliably.

1. Atomicity: This ensures that all operations within a transaction are treated as a single unit. Either all of them are executed successfully, or none are. If any part of the transaction fails, the entire transaction is rolled back (undone), maintaining data integrity.
2. Consistency: Consistency ensures that a transaction brings the database from one valid state to another. Integrity constraints are maintained so that the database remains consistent before and after the transaction.
3. Isolation: Isolation ensures that transactions are securely and independently processed at the same time without interference, but the results of the transaction are as if the transactions were processed sequentially. This prevents transactions from reading intermediate (and possibly inconsistent) data.
4. Durability: Durability guarantees that once a transaction has been committed, it will remain so, even in the event of a system failure. This means that the changes made by the transaction are permanently stored in the database. In practical terms, this means that the database system has mechanisms in place, such as writing to a transaction log, that ensure the permanence of the transaction's effects.

Together, these properties ensure that database transactions are executed safely, reliably, and in a way that preserves the integrity of the database.

**Best Practices**

- Short and Concise: Keep transactions as brief as possible.
- Error Handling: Use `TRY...CATCH` for robust error handling.
- Testing: Always test transactions thoroughly in a non-production environment.

**Conclusion**

Transactions are fundamental in ensuring data integrity, especially in scenarios like bank transfers. They provide a way to group multiple operations into a single, atomic unit, ensuring that either all operations succeed or none do, thus maintaining the consistency and reliability of your database.

This lesson now accurately represents the concept and implementation of transactions in T-SQL, particularly highlighting a practical example of a bank transfer.

## Example Performing a Bank Transfer

اعمل الجدول ده وبعدين تعالي نشوف المثال

```sql
-- Create Accounts Table
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    Balance DECIMAL(10, 2)
);


-- Create Transactions Table
CREATE TABLE Transactions (
    TransactionID INT PRIMARY KEY IDENTITY(1,1),
    FromAccount INT,
    ToAccount INT,
    Amount DECIMAL(10, 2),
    Date DATETIME
);


-- Insert Sample Data into Accounts
INSERT INTO Accounts (AccountID, Balance) VALUES (1, 500.00); -- Account 1
INSERT INTO Accounts (AccountID, Balance) VALUES (2, 300.00); -- Account 2
```

دلوقتي عندنا حساب رقم 1 فيه 500 وحساب رقم 2 فيه 300

عاوزين نسحب من اول حساب 100 دولار ونضيفها في الحساب رقم 2 ونسجل الtransaction دي بس يتنفذوا كلهم مع بعض

هنعملها ازاي؟

تعالي الأول نكتب الأوامر اللي عايزين ننفذها

```
update  Accounts
set Accounts.Balance=Accounts.Balance-100
where Accounts.AccountID=1

update  Accounts
set Accounts.Balance=Accounts.Balance-100
where Accounts.AccountID=1

insert into Transactions
(FromAccount,ToAccount,Amount,Date)
values
(1,2,100,GETDATE())
```

طيب تعالي نحطهم في transaction

عشان تعمل transaction بتعمل try catch  عاديه جدا بس بتزود فيها 3 حاجات

قبل ال try بتكتب begin transaction

جوه ال try بتحط الأوامر بتاعتك كلها واخر سطر بتكتب فيه commit

جوه ال catch بتكتب rollback

```
begin transaction
begin try

commit
end try
begin catch

rollback
end catch
```

فيصبح عندك الكود كده

```
begin transaction
begin try
update  Accounts
set Accounts.Balance=Accounts.Balance-100
where Accounts.AccountID=1
```

```
update  Accounts
set Accounts.Balance=Accounts.Balance+100
where Accounts.AccountID=2

insert into Transactions
(FromAccount,ToAccount,Amount,Date)
values
(1,2,100,GETDATE())
commit
end try
begin catch

rollback
end catch
select *FROM Accounts
```

هنا بقي بيقولك انك طالما ماعملتش commit الداتا عمرها ماهتتحط في الداتابيز وهتفضل في ال log file ولو ماعملتش rollback لو حصل خطأ مش هيرجع الداتا القديمه

كمان بيقولك انه الأفضل انك تحط في ال transaction الحاجات اللي ممكن الخطأ فيها يعملك مشكله فعلا لان الداتا بيز بتحجزلك الجدول مخصوص عشان ماحدش يعمل تعديل وانت شغال فهيفضل كل الناس مستنياك تخلص عشان يشتغلوا هما كمان

**Example: Performing a Bank Transfer**

**Setting Up the Database**

First, create two tables: `Accounts` and `Transactions`.

```
-- Create Accounts Table
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    Balance DECIMAL(10, 2)
);



-- Create Transactions Table
CREATE TABLE Transactions (
    TransactionID INT PRIMARY KEY IDENTITY(1,1),
```

```
    FromAccount INT,
    ToAccount INT,
    Amount DECIMAL(10, 2),
    Date DATETIME
);



-- Insert Sample Data into Accounts
INSERT INTO Accounts (AccountID, Balance) VALUES (1, 500.00); -- Account 1
INSERT INTO Accounts (AccountID, Balance) VALUES (2, 300.00); -- Account 2
```

### Example: Performing a Bank Transfer

We'll transfer $100 from Account 1 to Account 2 and record this transaction.

```
BEGIN TRANSACTION;



BEGIN TRY
    -- Subtract $100 from Account 1
    UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;



    -- Add $100 to Account 2
    UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;



    -- Log the transaction
    INSERT INTO Transactions (FromAccount, ToAccount, Amount, Date) VALUES (1, 2, 100,
GETDATE());



    -- Commit the transaction
    COMMIT;
END TRY
BEGIN CATCH
    -- Rollback in case of error
    ROLLBACK;
```

```
    -- Error handling code here
END CATCH;
```

In this script:

- The `BEGIN TRANSACTION` starts the transaction.
- `BEGIN TRY...END TRY` handles successful execution.
- `BEGIN CATCH...END CATCH` handles any errors, rolling back the transaction if necessary.
- `COMMIT` confirms the transaction; `ROLLBACK` undoes it in case of errors.

## Quiz

What is the definition of a transaction in SQL?

- A series of database operations that are treated as a single logical unit
- A way to group multiple tables in a database
- A single operation that updates a single row in a table
- A way to manipulate data in the database

Which of the following properties does a transaction in SQL adhere to?

- Atomicity, Consistency, Isolation, Durability
- Agility, Complexity, Inefficiency, Dependency
- Accuracy, Completeness, Integration, Durability
- Associativity, Commutativity, Idempotence, Distributability

Why are transactions important for Bank Transfers?

- To ensure that all operations are executed or none are
- To speed up the transfer process
- To manipulate the database structure
- To create backups of the database

What is the purpose of the BEGIN TRANSACTION statement in T-SQL?

- To start a transaction
- To end a transaction
- To log a transaction
- To rollback a transaction

Which statement is used to commit a transaction in T-SQL?

- COMMIT
- ROLLBACK
- BEGIN COMMIT
- END TRANSACTION

## Table Variables in T-SQL

هنا بيقولك انك تقدر تعمل متغير نوعه table وتقدر تعمل عليه أي عمليه انت عايزها وبيكون مؤقت غير الجداول العاديه بتاعت الداتابيز بيتمسح بمجرد خروجك من ال scope بتاعه

كمان بيكون اسرع من الphisical table لان ال sql server دايما بيراقبه وبيراقب العمليات اللي بتحصل عليه وكمان بيعمل عليه احصائيات

فيه منه نوعين table variable و temp table

ولما تستخدمه استخدمه في الحاجات البسيطه

بتقدر تعمل index واحد بس ومش بيعملها log ولا بيحسبلها statistics عشان يحسن من ادائها

## Introduction to Table Variables

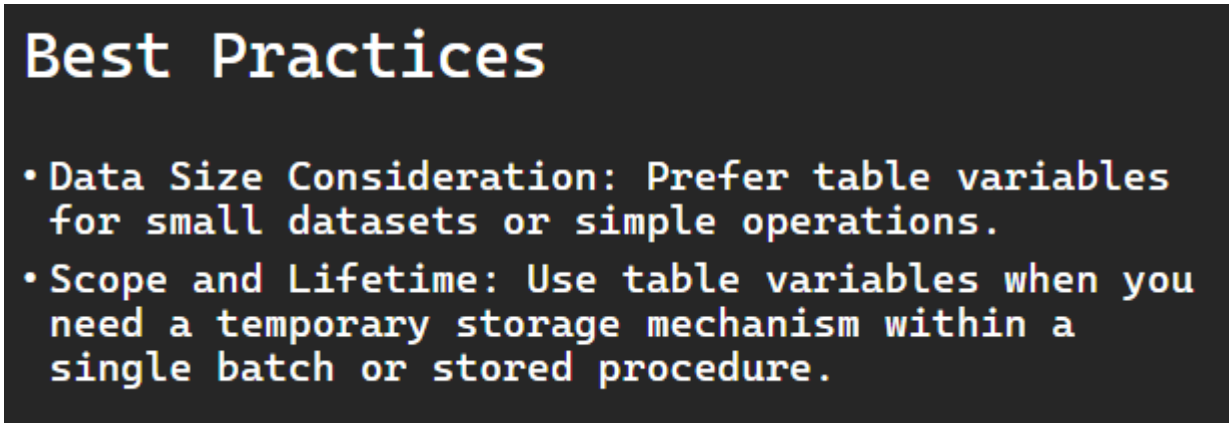
- Table variables in T-SQL are used to store a set of records temporarily, they have some distinct characteristics and are suitable for different scenarios.
- Table variables are declared using the DECLARE statement and are scoped to the batch, stored procedure, or function in which they are defined.

## Advantages of Table Variables

- Performance: For small datasets, table variables can be faster since they are stored in memory and not written to disk.
- Transaction Log: Operations on table variables generate fewer log records. This can be beneficial in terms of performance.
- Scope: The scope of a table variable is limited to the batch, stored procedure, or function in which it is defined. This can simplify transaction management and error handling.

## Limitations of Table Variables

- Indexing: By default, you can only create a primary key index at the time of declaration. Additional indexing options are limited.
- Statistics: Lack of statistics can lead to suboptimal query plans for large data sets.




## Table Variables in T-SQL

**Introduction to Table Variables**

Table variables in T-SQL are used to store a set of records temporarily, similar to temporary tables. However, they have some distinct characteristics and are suitable for different scenarios. Table variables are declared using the `DECLARE` statement and are scoped to the batch, stored procedure, or function in which they are defined.

**Advantages of Table Variables**

1. Performance: For small datasets, table variables can be faster since they are stored in memory and not written to disk.
2. Transaction Log: Operations on table variables generate fewer log records. This can be beneficial in terms of performance.
3. Scope: The scope of a table variable is limited to the batch, stored procedure, or function in which it is defined. This can simplify transaction management and error handling.

**Differences Between Table Variables and Temporary Tables**

- Logging and Transactions: Table variables have minimal logging for modifications, which can result in performance benefits for certain types of workloads. However, they don't participate fully in transactions. For example, if a transaction is rolled back, changes to a table variable made within that transaction are not rolled back.
- Statistics: SQL Server does not create statistics on table variables, which can affect the performance of queries involving large table variables.
- Scope: Temporary tables exist until they are explicitly dropped or the session/connection is closed, whereas table variables exist only within the batch, stored procedure, or function.

**Limitations of Table Variables**

1. Indexing: By default, you can only create a primary key index at the time of declaration. Additional indexing options are limited.
2. Statistics: Lack of statistics can lead to suboptimal query plans for large data sets.

**Best Practices**

- Data Size Consideration: Prefer table variables for small datasets or simple operations.
- Scope and Lifetime: Use table variables when you need a temporary storage mechanism within a single batch or stored procedure.

**Conclusion**

Table variables in T-SQL provide a convenient way to temporarily store and manipulate small sets of data. They are particularly useful for quick operations and in scenarios where minimal logging and transactional scope are important. Understanding when and how to use table variables, as opposed to temporary tables or other types of temporary storage, is an important skill in SQL programming and database design.

## Example

عشان تعرف variable table كأنك بتعرف متغير عادي بس بتفتح قوسين وتحط فيهم اسامي الاعمده

```
declare @EmployeesTable as table(
EmployeeID int,
Name nvarchar(50),
Department nvarchar(100)
)
```

وبتتعامل معاه كأنه جدول عادي

```
declare @EmployeesTable as table(
EmployeeID int,
Name nvarchar(50),
Department nvarchar(100)
)

insert into @EmployeesTable
values(1,'Mohammed','Marketing')
```

```sql
insert into @EmployeesTable (EmployeeId, Name, Department)
values (11, 'Ali', 'Sales');

SELECT * FROM @EmployeesTable WHERE Department = 'Sales';
select * from @EmployeesTable
```

## Quiz

What are table variables used for in T-SQL?

- Storing a set of records temporarily
- Creating primary key indexes
- Managing transactions
- Executing batch commands

What is the correct syntax to declare a table variable in T-SQL?

- CREATE TABLE @MyTable (...)
- DECLARE @MyTable TABLE (...)
- SET @MyTable = TABLE (...)
- DEFINE TABLE @MyTable (...)

Table variables are stored in which database?

- master
- temp
- They are stored in memory and not in a specific database

## What is the scope of a table variable?

- Global to the entire database.
- Limited to the session in which it was created.
- Limited to the batch, stored procedure, or function in which it is defined.
- Visible across multiple databases.

## Can you use a table variable in a JOIN clause?

- No, table variables cannot be used in JOINs.
- Yes, table variables can be used in JOINs like regular tables.
- Yes, but only with other table variables.

## What is a limitation of table variables in T-SQL?

- Limited memory usage
- Limited indexing options
- Limited transactional scope
- Limited execution speed

## How are changes to table variables handled in transactions?

- Changes are not logged, and cannot be rolled back.
- Changes are fully logged and can be rolled back.

What is an advantage of using table variables in terms of performance?

- Faster execution for large datasets
- Reduced memory usage
- Minimal logging for modifications
- Automatic creation of statistics

Can table variables be indexed in T-SQL?

- No, table variables cannot be indexed.
- Yes, but only at the time of declaration as a primary key.
- Yes, they can be indexed at any time like regular tables.
- Yes, but only non-clustered indexes are allowed.

Which of the following statements is true about table variables?

- They are automatically visible to all sessions and connections.
- They persist after the session that created them ends.
- They are cleaned up automatically at the end of the batch, procedure, or function.
- They require an explicit DROP statement to be deleted.

## Introduction to Temporary Tables

ال temp table زيه زي ال variable table بس الفرق انه ال variable table بيكون في ال memory انما ده بيتعمل فعلا في الداتابيز بس مش في الداتا بيز بتاعتك لا في داتابيز اسمها temp وتقدر برضه تعمل عليه أي عمليات ولو نسيت تحذفه هوا بيتحذف لوحده

وفيه منه نوعين:-

1- local temporary table :- وطريقته انك تكتب # وبعدها create table وبعدين اسم الجدول وده بيكون متاح فقط للconnection اللي عمله ولو ال connection انقطع بيتحذف

2- global temporary table :- وده نفس الطريقه بتاعت اللي فات بس بتزود # فكرته انه بيكون متاح لكل ال connections وبيتحذف بعد ما اخر connection ينقطع

كده كده هوا بيتحذف تلقائي بس يفضل انك تحذفه بال drop table

## Introduction to Temporary Tables

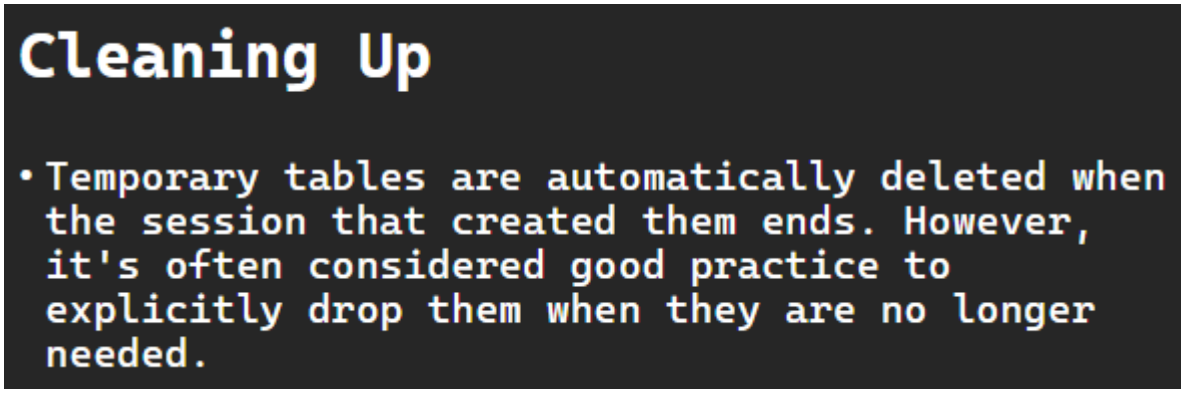
- Temporary tables in T-SQL are used to store and process intermediate results.
- These tables are created in the tempdb database and are automatically deleted when they are no longer used.
- Temporary tables are particularly useful in complex SQL operations where intermediate results need to be stored temporarily.

## Advantages of Temp Tables.

- Performance: Can improve performance in complex queries by breaking them into simpler parts.
- Complex Data Processing: Useful for storing intermediate results in complex data processing.
- Transaction Management: Changes in a temporary table are not logged extensively, which can be beneficial in large transactions.

## Types of Temporary Tables

- Local Temporary Tables: Created with a single hash (#) symbol. Visible only to the connection that creates it and are deleted when the connection is closed.
  - Syntax: CREATE TABLE #TempTable (...)
- Global Temporary Tables: Created with a double hash (##) symbol. Visible to all connections and are deleted when the last connection using it is closed.
  - Syntax: CREATE TABLE ##TempTable (...)




**Introduction to Temporary Tables**

Temporary tables in T-SQL are used to store and process intermediate results. These tables are created in the tempdb database and are automatically deleted when they are no longer used. Temporary tables are particularly useful in complex SQL operations where intermediate results need to be stored temporarily.

**Types of Temporary Tables**

1. Local Temporary Tables: Created with a single hash (`#`) symbol. Visible only to the connection that creates it and are deleted when the connection is closed.
2. Syntax: `CREATE TABLE #TempTable (...)`
3. Global Temporary Tables: Created with a double hash (`##`) symbol. Visible to all connections and are deleted when the last connection using it is closed.
4. Syntax: `CREATE TABLE ##TempTable (...)`

**Advantages of Temporary Tables**

1. Performance: Can improve performance in complex queries by breaking them into simpler parts.
2. Complex Data Processing: Useful for storing intermediate results in complex data processing.
3. Transaction Management: Changes in a temporary table are not logged extensively, which can be beneficial in large transactions.

**Cleaning Up**

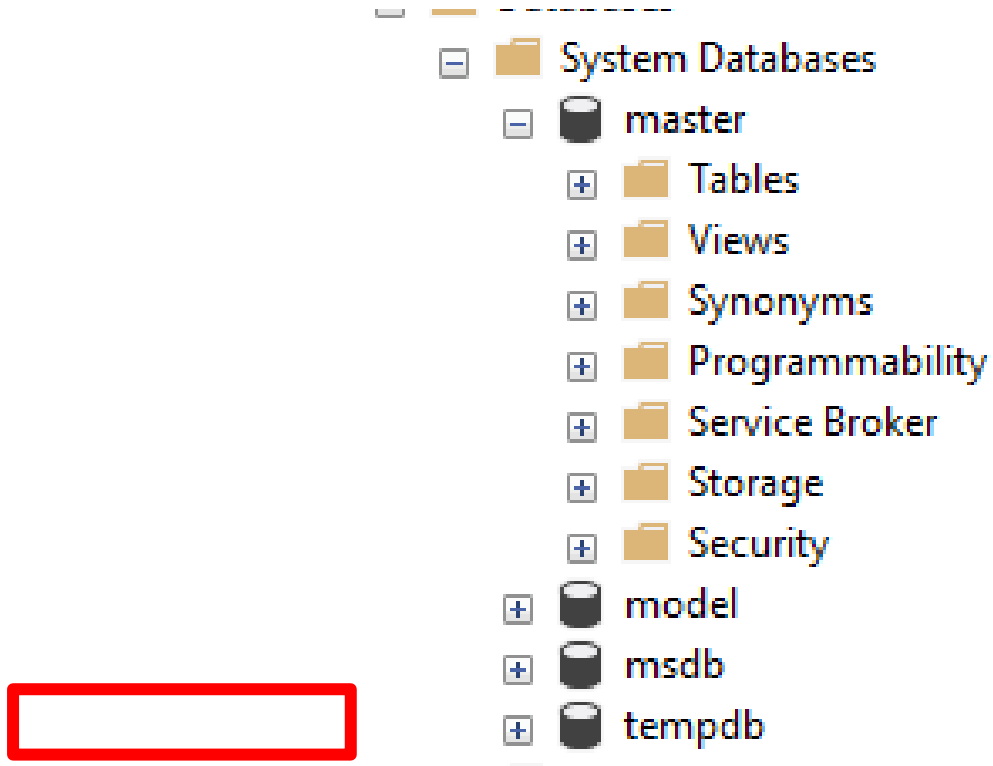
Temporary tables are automatically deleted when the session that created them ends. However, it's often considered good practice to explicitly drop them when they are no longer needed.

**Conclusion:**

Temporary tables are a powerful feature in T-SQL, allowing for efficient handling of complex queries and data processing tasks. Their ability to store intermediate results and their scope of visibility make them a versatile tool for database developers and administrators. Understanding when and how to use temporary tables can significantly optimize SQL operations.

## Example

ده مكان ال temdb بس ماتلعبش فيه




ودي طريقة التعامل بيه مش محتاجه شرح

```
-- Create a local temporary table named #EmployeesTemp
-- This table will be stored in the tempdb database and is visible only to this session
CREATE TABLE #EmployeesTemp (
	EmployeeId INT,
	Name VARCHAR(100),
	Department VARCHAR(50)
);

-- Insert a records into the #EmployeesTemp table
INSERT INTO #EmployeesTemp (EmployeeId, Name, Department)
VALUES (10, 'Mohammed', 'Marketing');

INSERT INTO #EmployeesTemp (EmployeeId, Name, Department)
VALUES (11, 'Ali', 'Sales');

-- Query the table
SELECT * FROM #EmployeesTemp WHERE Department = 'Sales';

-- Drop (delete) the temporary table #EmployeesTemp
-- This is a good practice to clean up, although the table would automatically be deleted
-- when the session ends
DROP TABLE #EmployeesTemp;
```

## Introduction to Temporary Tables

Temporary tables in T-SQL are used to store and process intermediate results. These tables are created in the tempdb database and are automatically deleted when they are no longer used. Temporary tables are particularly useful in complex SQL operations where intermediate results need to be stored temporarily.

## Types of Temporary Tables

1. Local Temporary Tables: Created with a single hash (`#`) symbol. Visible only to the connection that creates it and are deleted when the connection is closed.
2. Syntax: `CREATE TABLE #TempTable (...)`
3. Global Temporary Tables: Created with a double hash (`##`) symbol. Visible to all connections and are deleted when the last connection using it is closed.
4. Syntax: `CREATE TABLE ##TempTable (...)`

## Advantages of Temporary Tables

1. Performance: Can improve performance in complex queries by breaking them into simpler parts.
2. Complex Data Processing: Useful for storing intermediate results in complex data processing.
3. Transaction Management: Changes in a temporary table are not logged extensively, which can be beneficial in large transactions.

**Example:**

```sql
-- Create a local temporary table named #EmployeesTemp
-- This table will be stored in the tempdb database and is visible only to this session
CREATE TABLE #EmployeesTemp (
    EmployeeId INT,
    Name VARCHAR(100),
    Department VARCHAR(50)
);

-- Insert a records into the #EmployeesTemp table
INSERT INTO #EmployeesTemp (EmployeeId, Name, Department)
VALUES (10, 'Mohammed', 'Marketing');

INSERT INTO #EmployeesTemp (EmployeeId, Name, Department)
VALUES (11, 'Ali', 'Sales');

-- Query the table
SELECT * FROM #EmployeesTemp WHERE Department = 'Sales';

-- Drop (delete) the temporary table #EmployeesTemp
-- This is a good practice to clean up, although the table would automatically be deleted
-- when the session ends
DROP TABLE #EmployeesTemp;
```

**Cleaning Up**

Temporary tables are automatically deleted when the session that created them ends. However, it's often considered good practice to explicitly drop them when they are no longer needed.

**Conclusion:**

Temporary tables are a powerful feature in T-SQL, allowing for efficient handling of complex queries and data processing tasks. Their ability to store intermediate results and their scope of visibility make them a versatile tool for database developers and administrators. Understanding when and how to use temporary tables can significantly optimize SQL operations.

**Quiz1**

## What are temporary tables used for?

- Storing and processing intermediate results
- Storing permanent data
- Optimizing complex joins
- Managing database transactions

## Where are temporary tables created?

- In the tempdb database
- In the master database
- In the user's default database
- In a separate schema

## What is the advantage of using temporary tables?

- Improved performance in complex queries
- Increased data security
- Simpler syntax in SQL operations
- Automatic data backup

How are local temporary tables different from global temporary tables?

- Local temporary tables are visible to all connections, while global temporary tables are visible only to the connection that creates them
- Local temporary tables are deleted when the connection is closed, while global temporary tables are deleted when the last connection using them is closed
- Local temporary tables require explicit dropping, while global temporary tables are automatically deleted
- Local temporary tables can be accessed from any database, while global temporary tables are restricted to the tempdb database

What is the recommended practice for cleaning up temporary tables?

- They are automatically deleted when the session ends, so no explicit cleanup is needed
- They should be explicitly dropped when they are no longer needed
- They should be transferred to permanent tables before closing the session
- They should be emptied but kept for future use

## Differences between Temp Table vs Variable Table

In T-SQL, which is the SQL server's extension for SQL, two common ways to store data temporarily are through temporary tables and table variables. Here's a lesson that outlines the differences between them:

- Definition and Scope:
    - Temporary Tables: Created using the `CREATE TABLE` statement, with the table name prefixed by `#` for local temporary tables (visible only in the current session) or `##` for global temporary tables (visible to all sessions). They are stored in the tempdb database.
    - Table Variables: Declared using the `DECLARE` statement and have a similar structure to permanent tables. The syntax is `DECLARE @TableName TABLE (column`

`definitions`). They have a limited scope and are typically used within the function, stored procedure, or batch in which they are declared.

- Lifetime:
    - Temporary Tables: Exist until they are explicitly dropped using the `DROP TABLE` command or until the session/connection that created them is closed.
    - Table Variables: Automatically cleaned up at the end of the batch, function, or stored procedure in which they are defined.
- Performance and Usage:
    - Temporary Tables: Suitable for larger datasets and complex operations, like joining with other tables. They support indexes, statistics, and can result in better query performance for large data sets.
    - Table Variables: Better for smaller datasets and simpler operations. They have lower overhead but lack some of the optimizations available to temporary tables, like precompiled execution plans and statistics.
- Transaction Logs:
    - Temporary Tables: Fully logged in the transaction log, which can impact performance for large data manipulation operations.
    - Table Variables: Have minimal logging and do not participate in transactions. This means that if a transaction is rolled back, changes made to a table variable within that transaction are not rolled back.
- Use Cases:
    - Temporary Tables: Ideal for complex operations, temporary storage of data that requires rollback capabilities, and when working with a large number of rows.
    - Table Variables: Useful for quick, temporary storage of a small amount of data that does not require transactional rollbacks or heavy-duty operations.

Understanding when to use temporary tables versus table variables is crucial for optimizing performance and resource utilization in SQL Server databases.

**Quiz2**

## Which statement is used to create temporary tables in T-SQL?

- CREATE TEMP TABLE
- CREATE TABLE
- DECLARE TABLE
- CREATE TEMPORARY TABLE

## What is the scope of temporary tables in T-SQL?

- Limited to the current session
- Visible to all sessions
- Limited to the current batch
- Visible to all batches

## How long do temporary tables exist in T-SQL?

- Until explicitly dropped
- Until the session/connection is closed
- Until the end of the batch
- Until the end of the transaction

Which type of table is better for larger datasets and complex operations in T-SQL?

- Temporary tables
- Table variables
- Both are equally suitable
- Neither is suitable for larger datasets

Which type of table has minimal transaction logging in T-SQL?

- Temporary tables
- Table variables
- Both have the same transaction logging
- Neither has minimal transaction logging

When should table variables be used in T-SQL?

- For quick, temporary storage of a large amount of data
- For complex operations requiring rollback capabilities
- For quick, temporary storage of a small amount of data
- For heavy-duty operations requiring transactional rollbacks

## Differences between temporary tables and normal (permanent) tables

# Differences between temporary tables and normal (permanent) tables:

The differences between temporary tables and normal (permanent) tables in SQL are significant in terms of scope, lifespan, usage, and physical storage. Here's a breakdown of the key differences:

## 1. Lifespan and Scope

- Temporary Tables: They are created in the tempdb database and exist only for the duration of the session or connection that created them. Local temporary tables (prefixed with `#`) are visible only to the connection that created them, while global temporary tables (prefixed with `##`) are visible to all connections but still exist only until the last connection using them is closed.
- Normal Tables: Permanent tables are created in a user-defined database and persist until they are explicitly dropped by a user. They are visible and accessible to any user with the appropriate permissions, regardless of the user session or connection.

## 2. Performance and Storage

- Temporary Tables: They are stored in the tempdb database, which is a system database recreated every time SQL Server restarts. Operations on temporary tables generally have less logging and lower locking overhead, which can lead to performance benefits, especially for complex queries and large data manipulations.
- Normal Tables: Permanent tables are stored in the database in which they are created and are subject to more extensive logging and locking. This ensures data integrity and durability, which are critical for persistent data storage.

## 3. Usage

- Temporary Tables: Ideal for storing intermediate results in complex queries, for data processing within stored procedures, and for situations where data needs to be isolated to a single session or connection.
- Normal Tables: Used for storing data that needs to persist beyond the current session, for shared access among multiple users, and for data that forms the core structure of the application’s database schema.

## 4. Transaction Logging

- Temporary Tables: They have minimal transaction logging. This means that while they do participate in transactions, rollbacks and other transactional controls might have less overhead compared to normal tables.
- Normal Tables: Fully participate in transactions with complete logging, ensuring data integrity and supporting complex transactional controls.

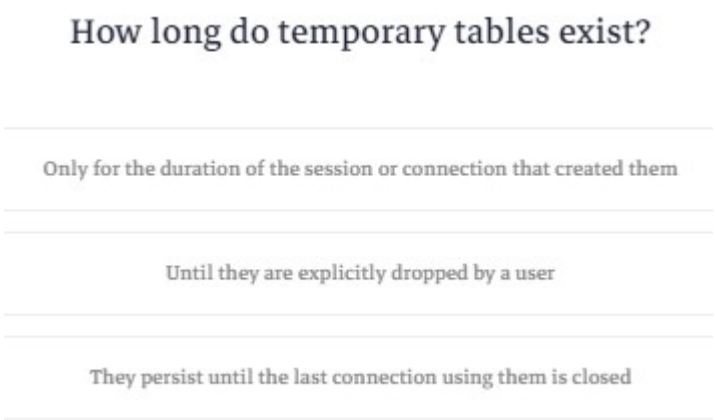
## 5. Backup and Recovery

- Temporary Tables: They are not included in database backups and cannot be recovered after a server restart or crash.
- Normal Tables: They are included in database backups and can be recovered in case of server restarts or database failures.

## Conclusion

Choosing between temporary and normal tables depends on the specific requirements of the task at hand. Temporary tables are ideal for transient data and quick, session-specific operations, whereas normal tables are suited for storing persistent data that requires full transactional support, backup, and recovery.

**Quiz3**



### Where are temporary tables stored?

- In the user-defined database
- In the tempdb database
- In the database in which they are created

### What is the main usage of temporary tables?

- Storing persistent data for shared access
- Storing intermediate results in complex queries
- Storing data that forms the core structure of the application's database schema

### Do temporary tables participate in transactions?

- No, they do not participate in transactions
- Yes, but with less transaction logging compared to normal tables
- Yes, with complete transactional support

### Are temporary tables included in database backups?

- Yes, they are included in database backups
- No, they are not included in database backups
- They can be recovered in case of server restarts or database failures

## Stored Procedures in T-SQL

علي مستوي ال tsql تقدر تعمل stored procedure ياما تعمل function

ال stored procedure هيا method تقدر تكتب فيها أي كود وهيا اسرع من انك تكتب الكود بتاعك في الشارع لانها بتكون pre compiled بتاخد وقت بس عبال ماتشغلها اول مره وبعد كده بتشتغل علي طول

كمان انك تبعت للداتا بيز اسم ال stored procedure وهوا يشغلها ده بيخلي ال network traffic اقل من انك تبعت الكود كله وكمان بيكون أأمن

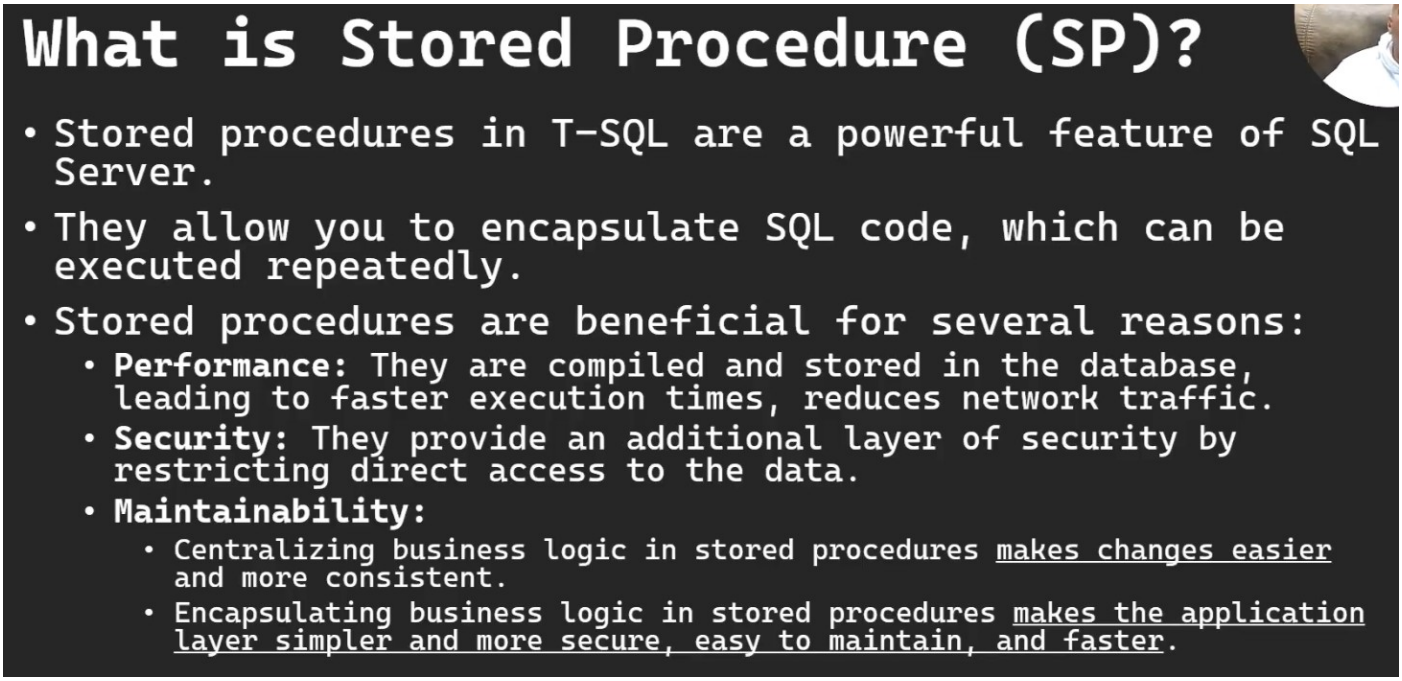
وتقدر تكتب فيها أي حاجه حتي انك ممكن تكتب ال query بتاعتك في شكل string وبعدين تنفذها بس ده غير مفضل وتقدر تنادي stored procedures تانيه او functions

تقدر كمان تنادي web service وتجيب الداتا بتاعتك منها وتقدر ترجع اكتر من result set وتقدر تعمل encryption للكود بتاعك

الفرق بينه وبين ال function انه ال function تقدر تستخدمها جوه ال select statement

طب هل ال stored procedure هوا case sensitive ؟

قالك ده بيعتمد علي ال configuration اللي انت عاملها وقت مانزلت ال sql server

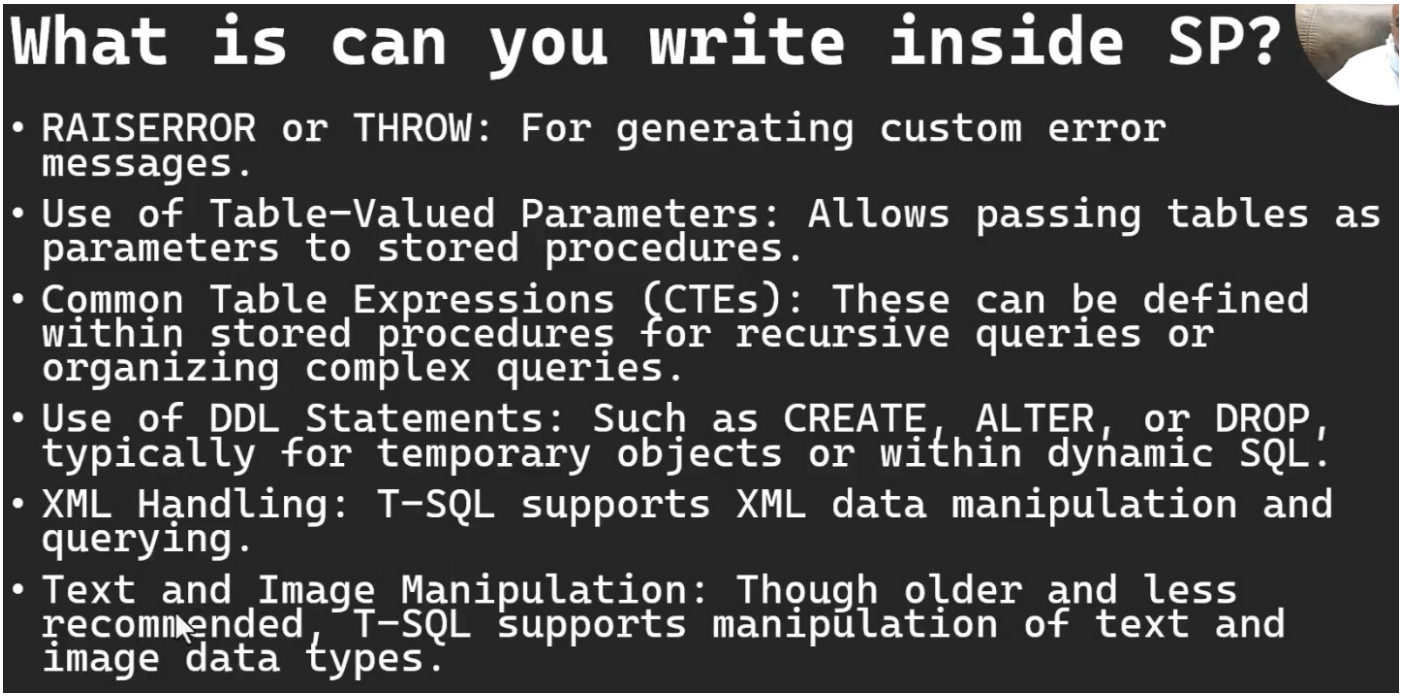


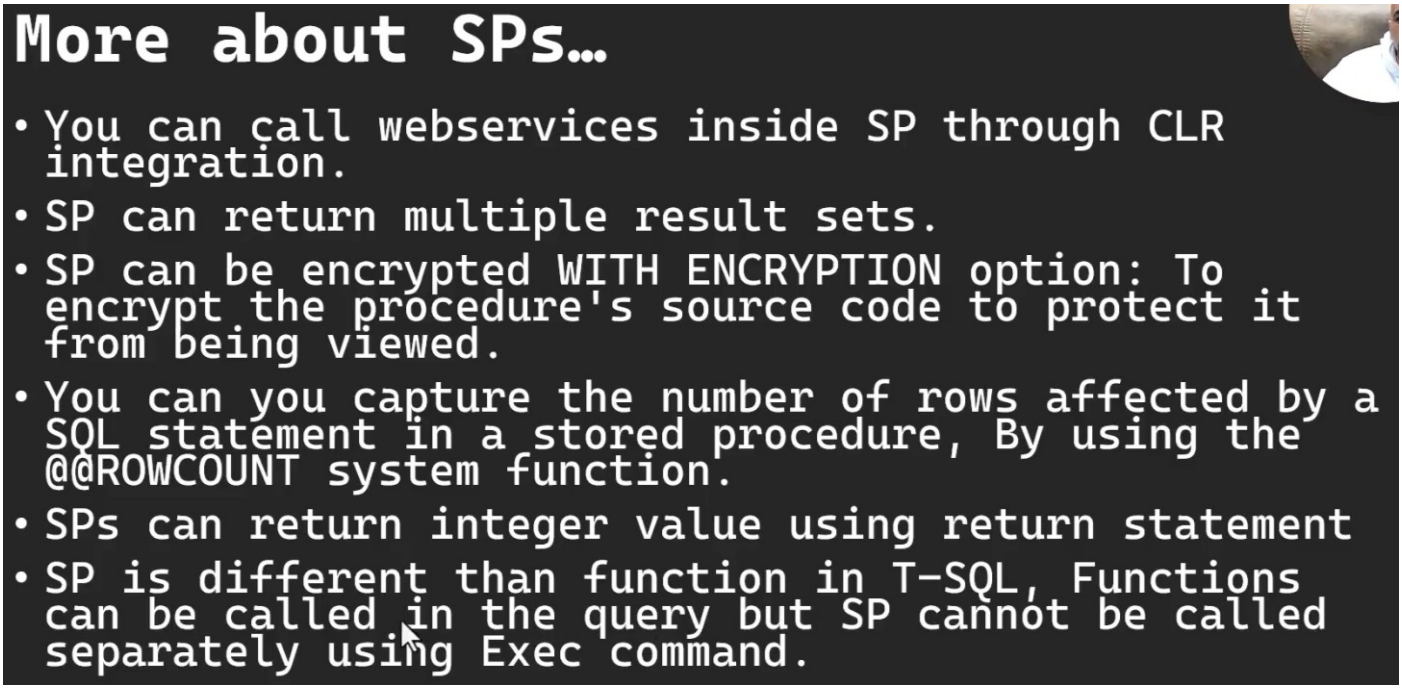
# What is can you write inside SP?

- SQL Queries and DML Statements: This includes SELECT, INSERT, UPDATE, DELETE, and MERGE statements for data querying and manipulation.
- Variable Declarations and Assignments: You can declare local variables using the DECLARE statement and set values with the SET or SELECT statements.
- Control Flow Statements:
  - IF...ELSE: For conditional logic.
  - WHILE: For looping.
  - BEGIN...END: To define blocks of code.
  - WAITFOR: To delay execution.
  - GOTO: For jumping to a labeled point in the procedure (though generally discouraged due to readability concerns).
- Error Handling and Transactions:
  - TRY...CATCH: For catching and handling exceptions.
  - TRANSACTION Management: Using BEGIN TRANSACTION, COMMIT, and ROLLBACK to handle transactions.

# What is can you write inside SP?

- Dynamic SQL Execution: Using EXEC or sp_executesql to execute dynamically built SQL strings.
- Calling Other Stored Procedures and Functions: You can call other stored procedures or user-defined functions within a stored procedure.
- Temporary Tables and Table Variables: You can create and use temporary tables and table variables for intermediate data storage and manipulation.
- Cursor Management: Although generally less efficient than set-based operations, cursors for row-by-row processing are supported in T-SQL.
- System Stored Procedures and Functions Calls: T-SQL allows calling system stored procedures and functions for various tasks.
- **Output Parameters**: Stored procedures can have output parameters to return data back to the caller.




# Stored Procedures in T-SQL

**Introduction to Stored Procedures**

Stored procedures in T-SQL are a powerful feature of SQL Server. They allow you to encapsulate SQL code, which can be executed repeatedly. Stored procedures are beneficial for several reasons:

- Performance: They are compiled and stored in the database, leading to faster execution times.
- Security: They provide an additional layer of security by restricting direct access to the data.

- Maintainability: Centralizing business logic in stored procedures makes changes easier and more consistent.

**What can you write inside Stored Procedure:**

In T-SQL, which is the SQL language variant used by Microsoft SQL Server, stored procedures can contain a wide range of SQL statements, control structures, and special features. Here's a detailed list of what you can write inside stored procedures in T-SQL:

- SQL Queries and DML Statements: This includes `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements for data querying and manipulation.
- Variable Declarations and Assignments: You can declare local variables using the `DECLARE` statement and set values with the `SET` or `SELECT` statements.
- Control Flow Statements:
    - `IF...ELSE`: For conditional logic.
    - `WHILE`: For looping.
    - `BEGIN...END`: To define blocks of code.
    - `WAITFOR`: To delay execution.
    - `GOTO`: For jumping to a labeled point in the procedure (though generally discouraged due to readability concerns).
- Error Handling and Transactions:
    - `TRY...CATCH`: For catching and handling exceptions.
    - `TRANSACTION` Management: Using `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` to handle transactions.
- Dynamic SQL Execution: Using `EXEC` or `sp_executesql` to execute dynamically built SQL strings.
- Calling Other Stored Procedures and Functions: You can call other stored procedures or user-defined functions within a stored procedure.
- Temporary Tables and Table Variables: You can create and use temporary tables and table variables for intermediate data storage and manipulation.
- Cursor Management: Although generally less efficient than set-based operations, cursors for row-by-row processing are supported in T-SQL.

- System Stored Procedures and Functions Calls: T-SQL allows calling system stored procedures and functions for various tasks.
- Output Parameters: Stored procedures can have output parameters to return data back to the caller.
- RAISERROR or THROW: For generating custom error messages.
- Use of Table-Valued Parameters: Allows passing tables as parameters to stored procedures.
- Common Table Expressions (CTEs): These can be defined within stored procedures for recursive queries or organizing complex queries.
- Use of DDL Statements: Such as `CREATE`, `ALTER`, or `DROP`, typically for temporary objects or within dynamic SQL.
- XML Handling: T-SQL supports XML data manipulation and querying.
- Text and Image Manipulation: Though older and less recommended, T-SQL supports manipulation of text and image data types.

It's important to use best practices while writing stored procedures in T-SQL, such as avoiding unnecessary cursors, ensuring proper error handling, and preventing SQL injection when using dynamic SQL. The capabilities and syntax may evolve with different versions of SQL Server, so always refer to the specific version's documentation for the most accurate information.

## CREATE: Add a New Person SP.

عاوزين دلوقتي نعمل stored procedure هنعملها ازاي ؟

طيب بالراحه كده احنا عارفين انه ال stored procedure هوا زيه زي ال method صح ؟

صح

طيب أي method في الدنيا بتحتاج ايه؟

بتحتاج parameters وreturn type وكود واسم

طيب عشان نعمل stored procedure جديد بنكتب الامر create procedure

بعدين هنكتب اسمه

بعدها هنكتب ال paramters اللي ال stored procedure هتاخدها وده هتبقي كأنك بتعرف متغير عادي بس من غير كلمة declare

بعد كده ال return type وده بيكون متغير عادي بس بتكتب بعديه كلمة output

كده فاضل ال implementation

هنكتب كلمة as وبعدها begin و end وفي وسطهم الكود اللي احنا عايزينه

ونشغل الكود هنلاقيها اتحفظت معانا

تعالي نعمل جدول ال people الأول

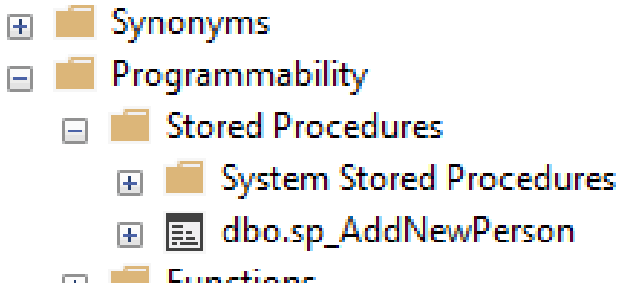
```
CREATE TABLE People (
   PersonID INT IDENTITY(1,1) PRIMARY KEY,
   FirstName NVARCHAR(100),
   LastName NVARCHAR(100),
   Email NVARCHAR(255)
);
```

ده الكود بتاع ال stored procedure

هنعمل stored procedure تضيف person جديد للجدول في كل مره هنشغلها فيها

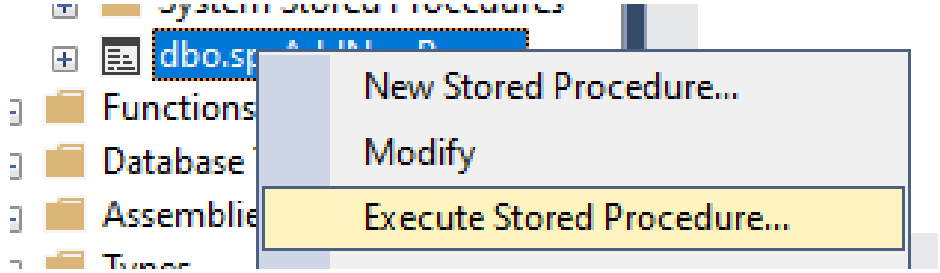
```
create Procedure sp_AddNewPerson
@FirstName nvarchar(100),
@LastName nvarchar(100),
@Email nvarchar(255),
@NewPersonID int output

as

begin
insert into People
(FirstName,LastName,Email)
values(@FirstName,@LastName,@Email)

set @NewPersonID=SCOPE_IDENTITY()

end
```

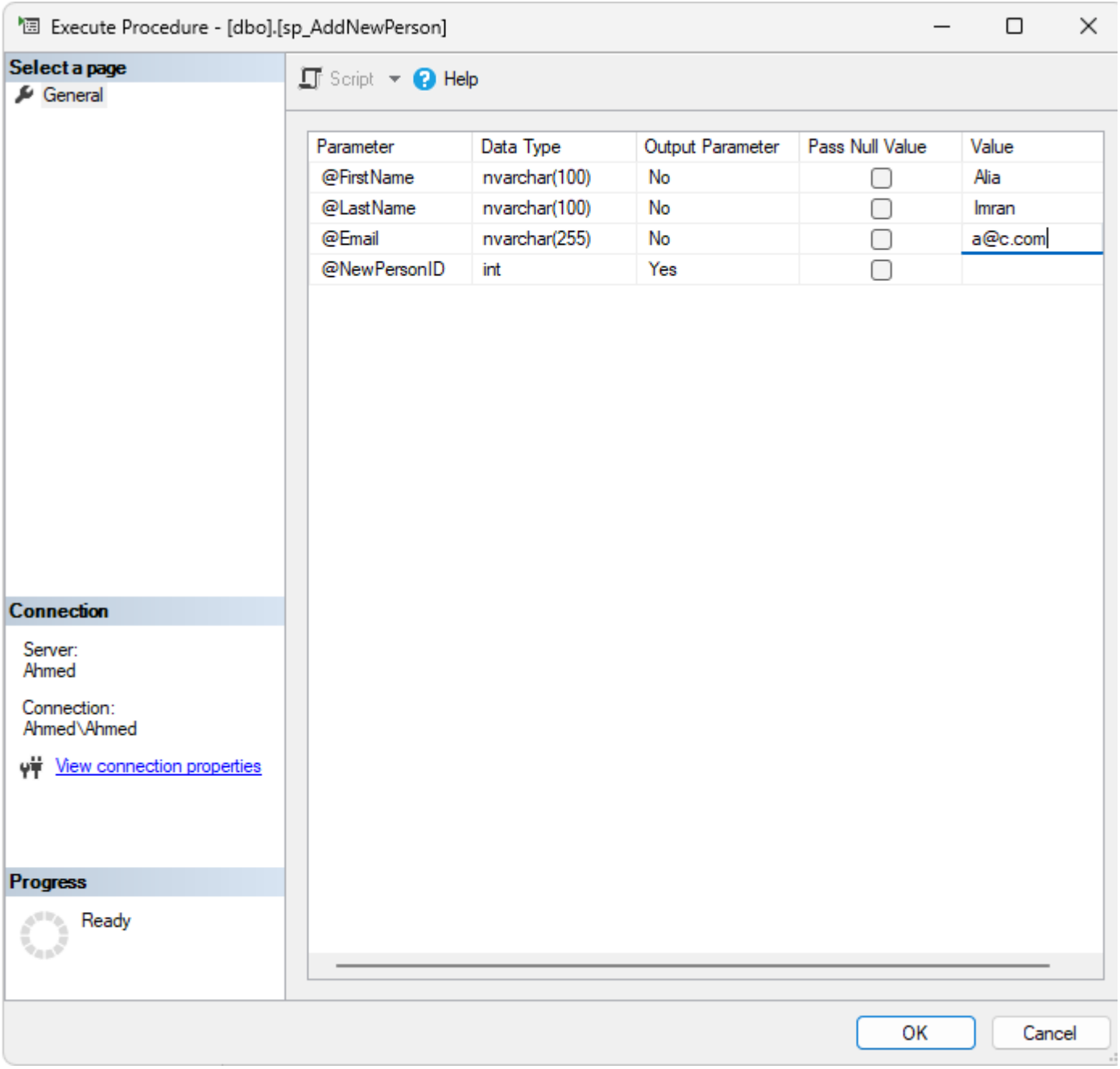
وده المكان اللي اتحطت فيه ال STORED PROCEDURE



طيب عشان نشغلها عندي طريقتين

لو طريقه عن طريق الماوس بيفتحلي شاشه بكتب فيها القيم بتاعت ال PARAMETERS بايدي




وبعدين هلاقيه طلعلي ال script ده ونفذه

```
USE [C21_DB1]
GO
```

```
DECLARE @return_value int,
        @NewPersonID int

EXEC@return_value = [dbo].[sp_AddNewPerson]
        @FirstName = N'Alia',
        @LastName = N'Imran',
        @Email = N'a@c.com',
        @NewPersonID = @NewPersonID OUTPUT

SELECT  @NewPersonID as N'@NewPersonID'

SELECT  'Return Value' = @return_value

GO
```

الطريقه التانيه عن طريق الكود

هنعرف متغير الأول عشان نخزن فيه الداتا اللي راجعه

وبعدين هنكتب الامر exec وبعديه اسم ال procedure وبعدين نديه القيم اللي طالبها

بالنسبه لل return هنقوله ان ال id هيتخزن في المتغير اللي عملناه

```
declare @PersonID int

exec sp_AddNewPerson
@FirstName='John',
@LastName='Doe',
@Email='john.doe@example.com',
@NewPersonID=@PersonID output

SELECT @PersonID AS NewPersonID;

select * from People
```

عشان تعدل عال stored procedure بدل create بتكتب alter

## CREATE: Add a New Person SP.

Note: always use prefix SP_ .

```
CREATE PROCEDURE SP_AddNewPerson
```

```
    @FirstName NVARCHAR(100),
    @LastName NVARCHAR(100),
    @Email NVARCHAR(255),
    @NewPersonID INT OUTPUT
AS
BEGIN
    INSERT INTO People (FirstName, LastName, Email)
    VALUES (@FirstName, @LastName, @Email);



    SET @NewPersonID = SCOPE_IDENTITY();
END
```

Creating a stored procedure named `SP_AddNewPerson` in SQL Server. This stored procedure is designed to add a new person's record to a table named `People` and return the autogenerated ID (the identity column value) of the newly added record. Here's a breakdown of each part of the script:

- Creating the Stored Procedure: `CREATE PROCEDURE SP_AddNewPerson` begins the definition of a new stored procedure named `SP_AddNewPerson`.
- Parameters:
    - `@FirstName NVARCHAR(100)`: This parameter accepts the first name of the person as an `NVARCHAR` string, with a maximum length of 100 characters.
    - `@LastName NVARCHAR(100)`: This parameter accepts the last name, also as an `NVARCHAR` string with a maximum length of 100 characters.
    - `@Email NVARCHAR(255)`: This parameter accepts the email address, allowing a string length of up to 255 characters.
    - `@NewPersonID INT OUTPUT`: This is an output parameter of type `INT`. It is used to return the ID of the newly inserted person back to the caller.
- Procedure Body:
    - The `BEGIN ... END` block encloses the SQL statements that the stored procedure will execute.
    - `INSERT INTO People (FirstName, LastName, Email) VALUES (@FirstName, @LastName, @Email);`: This statement inserts a new row into the `People` table.

The values for the `FirstName`, `LastName`, and `Email` columns are taken from the procedure's input parameters.

- `SET @NewPersonID = SCOPE_IDENTITY();`: After the insert operation, this line sets the `@NewPersonID` output parameter to the value returned by `SCOPE_IDENTITY()`. The `SCOPE_IDENTITY()` function returns the last identity value inserted into an identity column in the same scope (i.e., the `PersonID` of the newly added person in the `People` table).

- Usage:
  - When this stored procedure is executed, it will add a new row to the `People` table and return the identity (auto-incremented `PersonID`) of the new row through the `@NewPersonID` output parameter.

To execute this stored procedure and retrieve the new person's ID, you would use a SQL command similar to the following:

```
DECLARE @PersonID INT;
EXEC SP_AddNewPerson
    @FirstName = 'John',
    @LastName = 'Doe',
    @Email = 'john.doe@example.com',
    @NewPersonID = @PersonID OUTPUT;

SELECT @PersonID AS NewPersonID;
```

This stored procedure is useful in scenarios where you need to know the ID of a record immediately after it's inserted, such as for further operations on the newly created record.

## Scripts:

first execute the script that creates people table.

**READ: Get All People**

دلوقتي بقي اعملي procedure يرجعلي كل ال people

```
create Procedure sp_GetAllPeople
as
begin
```

```
select * from People
end
```

وده عشان نشغله

```
exec sp_GetAllPeople
```

## READ: Get All People

```
CREATE PROCEDURE SP_GetAllPeople
AS
BEGIN
    SELECT * FROM People
END
```

**To execute the `SP_GetAllPeople` stored procedure**, which you've defined to select and return all records from the `People` table, you can use a simple `EXEC` (execute) command in T-SQL. This command tells SQL Server to run the named stored procedure. Here's how you would write and execute the script:

```
EXEC SP_GetAllPeople;
```

This command will execute the `SP_GetAllPeople` stored procedure, which in turn runs the SQL query `SELECT * FROM People`. As a result, it will return all rows from the `People` table, including all columns (`PersonID`, `FirstName`, `LastName`, and `Email`).

Remember, when executing this command, you need to be connected to the database that contains the `People` table and the `SP_GetAllPeople` stored procedure.

## READ: Get Person By ID way 1

عاوزين نرجع شخص واحد من الداتابيز وهنا بيقولك ليها طريقتين واحده عن طريق ال stored procedure والتانيه عن طريق ال result set

هنعملها عن طريق ال stored procedure دلوقتي

```
create procedure sp_GetPersonByID
@PersonID int
```

```
as
begin
select * from People
where People.PersonID=@PersonID
end
```

وده عشان تشغلها

```
EXEC sp_GetPersonByID
@PersonID=1
```

## READ: Get Person By ID

```
CREATE PROCEDURE SP_GetPersonByID
    @PersonID INT
AS
BEGIN
    SELECT * FROM People WHERE PersonID = @PersonID
END
```

**Execute:**

```
EXEC SP_GetPersonByID
                @PersonID = 1
```

## READ: Get Person By ID way 2

هنرجع شخص برضه بس بالطريقه التانيه عن طريق ال output

الكود زيه زي ال #c بس اختلاف syntax مش اكتر

بس هوا بيفضل الطريقه الاولي لانها اسهل وكده كده السرعه تقريبا واحده

```
create procedure GetPersonByID2
@PersonID int,
@FirstName nvarchar(100)output,
@LastName nvarchar(100) output,
@Email nvarchar(255)output,
```

```
@IsFound bit output
as
begin
if exists(select 1 from People where
People.PersonID=@PersonID)
begin
select @FirstName=People.FirstName,
@LastName=People.LastName,
@Email=People.Email
from People
where People.PersonID=@PersonID

set @IsFound=1

end
else
begin
set @IsFound=0
end
end
```

```
declare @PersonID int=1
declare @FirstName nvarchar(100)
declare @LastName  nvarchar(100)
declare @Email nvarchar(255)
declare @IsFound bit

exec GetPersonByID2
@PersonID=@PersonID,
@FirstName=@FirstName OUTPUT,
@LastName=@LastName output,
@Email=@Email OUTPUT,
@IsFound=@IsFound OUTPUT

IF @IsFound=1
begin
 SELECT @FirstName as FirstName, @LastName as LastName,
@Email as Email;
end
else
```

```
PRINT 'Person not found';
```

another way is to create `SP_GetPersonByID2` stored procedure so that it retrieves a person's information as output parameters instead of a standard result set, you need to declare output parameters for each piece of information you want to retrieve. In this case, that would be `FirstName`, `LastName`, and `Email`.

If the person is not found in the database when using the `SP_GetPersonByID` stored procedure, you can include an additional output parameter that indicates whether a record was found. This parameter can be a boolean or an integer flag (often used in SQL Server).

**Here's the stored procedure:**

```
CREATE PROCEDURE SP_GetPersonByID2
    @PersonID INT,
    @FirstName NVARCHAR(100) OUTPUT,
    @LastName NVARCHAR(100) OUTPUT,
    @Email NVARCHAR(255) OUTPUT,
    @IsFound BIT OUTPUT  -- Additional parameter to indicate if a record was found
AS
BEGIN
    IF EXISTS(SELECT 1 FROM People WHERE PersonID = @PersonID)
    BEGIN
        SELECT
            @FirstName = FirstName,
            @LastName = LastName,
            @Email = Email
        FROM People
        WHERE PersonID = @PersonID;


        SET @IsFound = 1;  -- Set to 1 (true) if a record is found
    END
    ELSE
    BEGIN
        SET @IsFound = 0;  -- Set to 0 (false) if no record is found
    END
```

```
END
```

In this version:

- `@IsFound` is an output parameter of type `BIT` (which is essentially a boolean in SQL Server).
- The `IF EXISTS` statement checks if a record with the specified `PersonID` exists in the `People` table.
- If a record is found, it retrieves the details and sets `@IsFound` to `1`.
- If no record is found, it sets `@IsFound` to `0`.

**To execute this stored procedure and check if a person was found:**

```
DECLARE @ID INT = 1;  -- Example PersonID
DECLARE @FName NVARCHAR(100);
DECLARE @LName NVARCHAR(100);
DECLARE @Email NVARCHAR(255);
DECLARE @Found BIT;



EXEC SP_GetPersonByID2
    @PersonID = @ID,
    @FirstName = @FName OUTPUT,
    @LastName = @LName OUTPUT,
    @Email = @Email OUTPUT,
    @IsFound = @Found OUTPUT;



IF @Found = 1
    SELECT @FName as FirstName, @LName as LastName, @Email as Email;
ELSE
    PRINT 'Person not found';
```

- After executing the stored procedure, you check the value of `@Found`.

- If `@Found` is `1`, the person was found, and you can retrieve their details.
- If `@Found` is `0`, it means no person with the given ID was found in the database.

## UPDATE: Update a Person's Details

هنعمل procedure عشان يعمل update لل person

```
create procedure sp_UpdatePerson
@PersonID INT,
@FirstName NVARCHAR(100),
@LastName NVARCHAR(100),
@Email NVARCHAR(255)

as
begin
update People
set FirstName=@FirstName,
LastName=@LastName,
Email=@Email
where PersonID=@PersonID

    end
```

```
exec sp_UpdatePerson
@PersonID = 1,
   @FirstName = 'UpdatedFirstName',
   @LastName = 'UpdatedLastName',
   @Email = 'updated.email@example.com';
```

## DELETE: Remove a Person

هنحذف شخص

```
CREATE PROCEDURE SP_DeletePerson
   @PersonID INT
AS
BEGIN
   DELETE FROM People WHERE PersonID = @PersonID
END

EXEC SP_DeletePerson
```

```
        @PersonID = 4
```

## RETURN Statement

عشان نرجع داتا من ال stored procedure ياما نعمل ال output parameter زي ماكنا بنعمل ياما نستخدم ال return statement

ال stored procedure بترجع int ممكن نستعملها في اننا نعرف عدد الصفوف اللي اتاثرت او العمليه تمت ولا لا او الشخص موجود ولا لا وهكذا

ال return هنا بتخرجك بره ال procedure وبتستخدمها عادي

```
CREATE PROCEDURE SP_CheckPersonExists
   @PersonID INT
AS
BEGIN
   IF EXISTS(SELECT * FROM People WHERE PersonID =
@PersonID)
      RETURN 1;  -- Person exists
   ELSE
      RETURN 0;  -- Person does not exist
END
```

```
DECLARE @Result INT;
EXEC @Result = SP_CheckPersonExists @PersonID = 123; --
Replace 123 with the actual PersonID


IF @Result = 1
   PRINT 'Person exists.';
ELSE
   PRINT 'Person does not exist.';
```

## Understanding Return Values in Stored Procedures in T-SQL

**Introduction**

In T-SQL, a stored procedure can return a value to the calling environment. This is an essential feature for communicating the success or failure of the procedure's execution or to provide a specific status code. Understanding how to use and handle return values is crucial for effective database programming.

**What is a Return Value?**

- A return value is an integer value that a stored procedure can return to the caller.
- It is primarily used to indicate success or failure (often with `0` for success and non-zero values for various error conditions or specific states).
- Also return used to exist from SP.

**Example using return value to check if the person exists or not.**

Stored procedure to check if a person exists in a `People` table, to check for the existence of a person based on a unique identifier, such as `PersonID`. Here's the a stored procedure:

```
CREATE PROCEDURE SP_CheckPersonExists
    @PersonID INT
AS
BEGIN
    IF EXISTS(SELECT * FROM People WHERE PersonID = @PersonID)
        RETURN 1;  -- Person exists
    ELSE
        RETURN 0;  -- Person does not exist
END
```

**Explanation:**

- It takes `@PersonID` as a parameter, which is used to identify the person in the `People` table.
- The `IF EXISTS` statement checks for the existence of a record in the `People` table where the `PersonID` matches the provided parameter.
- It returns `1` if the person exists and `0` if not.

To use this stored procedure, you would call it with a specific `PersonID` and check the return value:

```
DECLARE @Result INT;
EXEC @Result = SP_CheckPersonExists @PersonID = 123; -- Replace 123 with the actual PersonID



IF @Result = 1
```

```
    PRINT 'Person exists.';
ELSE
    PRINT 'Person does not exist.';
```

In this usage example, `@Result` will hold the return value of the stored procedure, indicating whether the specified person exists (`1`) or not (`0`).

**Best Practices**

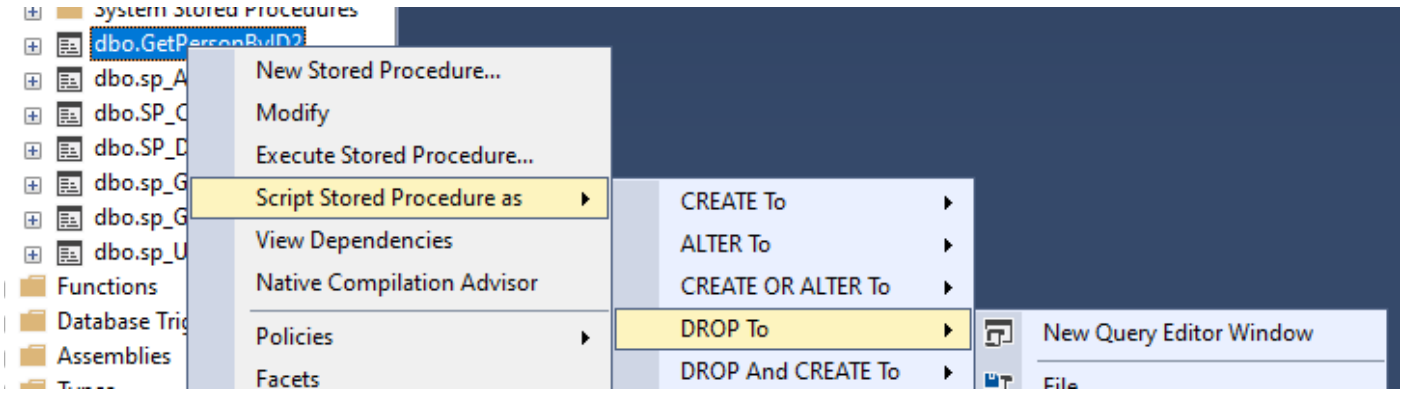
- Use return values primarily for indicating success or failure.
- For detailed data output, consider using OUTPUT parameters, which can return more complex data types.
- Clearly document the meaning of various return values in your stored procedures.

**Conclusion**

Return values in T-SQL stored procedures are a simple yet powerful way to communicate the outcome of a procedure. They are particularly useful for error handling and flow control in database applications. Understanding how to use them effectively is a key skill in T-SQL programming.

## Drop SP

عشان تحذف stored procedure




```
USE [C21_DB1]
GO

/****** Object:  StoredProcedure [dbo].[GetPersonByID2]
Script Date: 11/01/2024 6:28:50 PM ******/
DROP PROCEDURE [dbo].[GetPersonByID2]
GO
```

يااما كليك يمين وdelete

## SP_HELPTEXT command.

ال sp_helptext هوا stored procedure معمول جاهز تديله اسم procedure يجيبلك معلومات عنه

```
use C21_DB1;

EXEC sp_helptext 'SP_AddNewPerson';
```

The `sp_helptext` command in SQL Server is a system stored procedure that is used to retrieve the text definition of a stored procedure, function, trigger, view, or user-defined function in a SQL Server database. It is a useful tool for developers and database administrators to examine the source code or the SQL statements within these database objects. Here's a lesson on how to use the `sp_helptext` command:

Syntax:

```
sp_helptext [ @objname = ] 'object_name'
```

- `@objname`: The name of the stored procedure, function, trigger, view, or user-defined function whose text definition you want to retrieve. This parameter is of type `sysname`.

Usage:

1. Retrieving the Text Definition of a Stored Procedure:
2. To retrieve the text definition of a stored procedure, you can use the `sp_helptext` command as follows:

```
EXEC sp_helptext 'YourStoredProcedureName'
```

1. Replace `'YourStoredProcedureName'` with the name of the stored procedure you want to examine. This command will display the text definition of the stored procedure in the query results.
2. Retrieving the Text Definition of a Function, Trigger, View, or User-Defined Function:

3. You can use the `sp_helptext` command to retrieve the text definition of other database objects like functions, triggers, views, or user-defined functions in the same way as for stored procedures. Just replace `'YourObjectName'` with the name of the object you want to examine.

```
EXEC sp_helptext 'YourObjectName'
```

Example:

Let's say you have a stored procedure named `GetEmployeeInfo` that you want to examine. You can use `sp_helptext` as follows:

```
EXEC sp_helptext 'YourObjectName'
```

The result will display the SQL statements and code within the `GetEmployeeInfo` stored procedure.

Notes:

- The `sp_helptext` command is particularly useful when you want to view the source code of database objects for documentation, debugging, or understanding their logic.
- It only retrieves the text definition of a single database object at a time.
- Be cautious when using `sp_helptext` on sensitive database objects, as it exposes the underlying code. Ensure that you have the necessary permissions to execute this command.
- This command can be executed in SQL Server Management Studio (SSMS) or any other SQL query tool.

In summary, the `sp_helptext` command is a valuable tool for inspecting and examining the source code or SQL statements within stored procedures, functions, triggers, views, and user-defined functions in SQL Server databases. It helps developers and administrators gain insights into the logic and implementation of these database objects.

**Quiz**

## What is a stored procedure in T-SQL?

- A precompiled collection of SQL statements and optional control-of-flow statements.
- A single SQL statement.
- A special type of table in SQL Server.
- A database trigger.

## Which of the following is a benefit of using stored procedures?

- Reduced network traffic
- Increased network traffic
- Slower execution
- Harder maintenance

## Which SQL keyword is used to create a stored procedure?

- CREATE PROCEDURE
- MAKE PROCEDURE
- BUILD PROCEDURE
- SET PROCEDURE

## In SQL Server, are stored procedure names case-sensitive?

- Yes
- No
- It depends on the collation setting of the database or server. If the collation is case-sensitive (e.g., SQL_Latin1_General_CP1_CS_AS), stored procedure names are case-sensitive. If the collation is case-insensitive (e.g., SQL_Latin1_General_CP1_CI_AS), they are not case-sensitive.
- Case sensitivity for stored procedure names is determined at the time of SQL Server installation and cannot be changed later.

## How do you execute a stored procedure named 'GetEmployeeData'?

- RUN GetEmployeeData;
- EXEC GetEmployeeData;
- CALL GetEmployeeData;
- START GetEmployeeData;

## How can a stored procedure return a value?

- Using RETURN statement
- Using OUT parameter
- None of the above

## Can a stored procedure call another stored procedure?

- Yes, even if SP in another database
- No
- Only if they are in the same database

What is the purpose of the BEGIN and END statements in a stored procedure?

- To define the start and end of the procedure
- To declare variables
- To execute a query
- To comment code

Which of the following is used to pass a variable into a stored procedure?

- Input parameter
- Output parameter
- Local variable
- Global variable

What does the SCOPE_IDENTITY() function return?

- The last used identity value in the session
- The last used identity value in the current scope
- The first identity value in the table
- The total number of identity values in the table

Which statement is true about the OUTPUT parameters in a stored procedure?

- They are read-only.
- They cannot be used in a SELECT statement.
- They can return a value to the calling procedure.
- They are optional in all stored procedures.

What happens if you try to create a stored procedure with a name that already exists?

- The existing procedure is automatically deleted.
- The existing procedure is updated.
- Nothing happens to the existing SP, and An error is thrown.

Which command is used to modify an existing stored procedure?

- MODIFY PROCEDURE
- CHANGE PROCEDURE
- ALTER PROCEDURE
- UPDATE PROCEDURE

## Can a stored procedure return multiple result sets?

- Yes
- No
- Only if it does not contain any parameters
- Only in SQL Server 2019 and later

## Which of the following is a valid reason to use stored procedures?

- To prevent SQL injection attacks
- To directly modify the server's operating system
- To encapsulate business logic
- To improve performance

## Which of the following is a valid reason to use stored procedures?

- To prevent SQL injection attacks
- To directly modify the server's operating system
- To encapsulate business logic
- To improve performance

## Can stored procedures be used to generate dynamic SQL?

- Yes, using the EXECUTE statement
- No, it is not possible
- Only with a special permission
- Only if they are not returning any data

How can you improve the performance of a stored procedure?

- By adding more complex queries
- Using indexes effectively
- By increasing the number of parameters
- By calling more stored procedures inside it

What is parameter sniffing in the context of stored procedures?

- A method to improve security
- A process where SQL Server optimizes execution based on parameter values
- A type of SQL injection attack
- A debugging technique

What is the default value of a parameter if not specified in a T-SQL stored procedure?

- NULL
- 0
- Empty string
- It must be specified; there is no default.

Which T-SQL statement is used to delete a stored procedure?

- REMOVE PROCEDURE
- DELETE PROCEDURE
- DROP PROCEDURE
- ERASE PROCEDURE

What should be used to pass a table of data to a stored procedure?

- Table-valued parameter
- XML parameter
- CSV string
- CSV string

How can you prevent SQL injection in stored procedures?

- By using dynamic SQL
- By using parameterized queries
- By limiting the procedure to SELECT statements only
- By limiting the procedure to SELECT statements only

What is the purpose of the RETURN statement in a stored procedure?

- To return data to the caller
- To indicate the successful completion of the procedure
- To exit the procedure and return an integer status code
- To rollback transactions

How can the output of a stored procedure be captured?

- Using the OUTPUT keyword
- By redirecting to a file
- By using a SELECT statement at the end
- By using a temporary table

What is the main difference between a function and a stored procedure in SQL Server?

- A function cannot return a result set.
- A stored procedure can return multiple values, but a function can only return one.
- A function can be used in a SELECT statement, but a stored procedure cannot.
- A stored procedure is always faster than a function.

Can a stored procedure in T-SQL have a transaction within it?

- Yes, but only one transaction at a time
- Yes, including nested transactions
- No, transactions are not allowed
- Only if it is a read-only transaction

What happens when a stored procedure is called with the wrong number of parameters?

- It executes with default values for missing parameters.
- It does not execute and an error is returned.
- It guesses the missing parameters.
- It prompts the user for the missing parameters.

Can you use PRINT statements in a stored procedure for debugging purposes?

- Yes, and the output will be visible in the messages tab in SSMS
- No, PRINT statements are not allowed
- Yes, but the output will be stored in a log file
- Only if the debugging mode is enabled

What is the scope of variables declared in a stored procedure?

- Global within the database
- Local to the stored procedure
- Global within all databases on the server
- Local to all procedures within the same schema

What is the result of nesting too many stored procedures?

- Increased performance
- Potential stack overflow
- Automatic parallelization of queries
- Reduction in memory usage

Can you use Error Handling Try..Catch in SP?

- Yes
- No

What is the impact of using the sp_ prefix in the name of a user-defined stored procedure?

- It reserves the procedure for system use only.
- It improves the performance of the procedure.
- It may cause name conflict issues with system stored procedures.
- It automatically grants execute permissions to all users.

Which T-SQL command is used to view the definition of a stored procedure?

- DESCRIBE
- VIEW DEFINITION
- SHOW PROCEDURE
- SP_HELPTEXT

In a stored procedure, what is a good practice when handling errors?

- Ignore errors to avoid complexity
- Use TRY...CATCH blocks to handle exceptions
- Handle errors only at the application level
- Restart the SQL Server service on an error

What is a key benefit of encapsulating business logic in stored procedures?

- It increases the size of the database
- It makes the application layer simpler and more secure
- It is required by SQL Server
- It automatically optimizes all queries

How can you capture the number of rows affected by a SQL statement in a stored procedure?

- By using the @@ROWCOUNT system function.
- It is not possible to capture this information.
- By using the COUNT function.
- By using a special ROWS keyword.

## Can a stored procedure contain dynamic SQL?

- Yes, and it can be executed using EXEC or sp_executesql.
- No, stored procedures cannot contain dynamic SQL.
- Yes, but only in SQL Server Enterprise Edition.
- Only if the dynamic SQL is a SELECT statement.

## Can a stored procedure in T-SQL invoke a web service?

- Yes, directly through T-SQL commands
- No, T-SQL cannot interact with web services
- Yes, but only through CLR integration
- Only if the web service is hosted on the same server

## What is the use of the WITH ENCRYPTION option when creating a stored procedure?

- To compress the procedure for saving space
- To encrypt the procedure's source code to protect it from being viewed
- To enhance the performance of the procedure
- To encrypt data returned by the procedure

What is the role of the sp_executesql stored procedure in SQL Server?

To execute a batch of SQL statements repeatedly

To execute a dynamic SQL statement

To schedule the execution of a SQL statement

To execute a stored procedure within another stored procedure

You can you capture the number of rows affected by a SQL statement in a stored procedure, By using the @@ROWCOUNT system function.

True

False

## String Functions

هنا بيعرضلك ال built in functions الأكثر استخداما مع ال string

len بترجعلك طول ال string

upper بيحول النص لحروف كبيره

lower بيحول النص لحروف صغيرة

substring بيرجعلك جزء من النص وبتديله ال field وتقوله يبدأ من الحرف رقم كام وطول النص كام (الترقيم من واحد مش من صفر)

charindex بيدور عالحرف ويقولك هوا رقم كام

replace بيبدل كلمة مكان كلمه

concat بيدمج النصوص

left بيجيبلك عدد معين انت اللي بتحدده من الحروف من اليسار

right بيجيبلك عدد معين انت اللي بتحدده من الحروف من اليمين

ltrim لوفيه مسافات من يسار النص بيشيلها

rtrim لوفيه مسافات من يمين النص بيشيلها

trim لو فيه مسافات قبل او بعد النص بيشيلها

```sql
use C21_DB1;

--Common String Functions...

-- Using the LEN function to get the length of each employee's name
SELECT LEN(Name) AS NameLength FROM dbo.Employees2
GO

-- Converting employee names to uppercase using the UPPER function
SELECT UPPER(Name) AS UpperCaseName FROM dbo.Employees2
GO

-- Converting employee names to lowercase using the LOWER function
SELECT LOWER(Name) AS UpperCaseName FROM dbo.Employees2
GO

-- Extracting the first three characters of each name using SUBSTRING
SELECT SUBSTRING(Name, 1, 3) AS NameSubstring FROM dbo.Employees2
GO

-- Finding the position of '0' in each name using CHARINDEX
SELECT CHARINDEX('o', Name) AS CharPosition FROM dbo.Employees2
GO

-- Replacing 'Sales' with 'Marketing' in department names using REPLACE
SELECT REPLACE(Department, 'Sales', 'Marketing') AS NewDepartment FROM dbo.Employees2
GO

-- Concatenating the name and department with a hyphen in between using CONCAT
SELECT CONCAT(Name, ' - ', Department) AS ConcatenatedString FROM dbo.Employees2
GO

-- Practice Exercise: Format Name and Department in a specific format using CONCAT, UPPER, and LOWER
-- Objective: Format the Name and Department columns as a single string, where names are in
uppercase, and department names are in lowercase, separated by a colon (:)
SELECT CONCAT(UPPER(Name), ' : ', LOWER(Department)) AS FormattedOutput FROM dbo.Employees2
GO

-- Extracting the first 3 characters from the left side of the employee's name using LEFT
SELECT LEFT(Name, 3) AS LeftSubstring FROM dbo.Employees2
GO

-- Extracting the last 3 characters from the right side of the employee's name using RIGHT
SELECT RIGHT(Name, 3) AS RightSubstring FROM dbo.Employees2
GO

-- Removing leading spaces from the employee's name using LTRIM
SELECT LTRIM(Name) AS NameWithNoLeadingSpaces FROM dbo.Employees2
GO

-- Removing trailing spaces from the employee's name using RTRIM
SELECT RTRIM(Name) AS NameWithNoTrailingSpaces FROM dbo.Employees2
GO


-- Removing spaces from the start and end of each name using LTRIM and RTRIM
SELECT LTRIM(RTRIM(Name)) AS TrimmedName FROM dbo.Employees2
GO

-- Removing both leading and trailing spaces from the employee's name using TRIM
SELECT TRIM(Name) AS NameWithNoSpaces FROM dbo.Employees2
GO
```

# Overview of String Functions in T-SQL

In T-SQL, string functions are essential tools for manipulating and querying data in text format. Here's an overview of some commonly used string functions:

1. LEN

- Purpose: Returns the number of characters in a specified string, excluding trailing spaces.

2. UPPER

- Purpose: Converts all characters in a specified string to uppercase.

3. LOWER

- Purpose: Converts all characters in a specified string to lowercase.

4. SUBSTRING

- Purpose: Returns part of a string, starting at a specified position and for a specified length.

5. CHARINDEX

- Purpose: Returns the starting position of the first occurrence of a specified expression in a string.

6. REPLACE

- Purpose: Replaces all occurrences of a specified string value within a given string with another string value.

7. LTRIM

- Purpose: Removes leading spaces from a string.

8. RTRIM

- Purpose: Removes trailing spaces from a string.

9. CONCAT

- Purpose: Concatenates two or more strings into one string.

10. LEFT

- Purpose: Returns the left part of a string with the specified number of characters.

11. RIGHT

- Purpose: Returns the right part of a string with the specified number of characters.

12. TRIM

- Purpose: Removes both leading and trailing spaces from a string.

These functions are fundamental in processing and analyzing text data in SQL Server. Understanding and applying them appropriately can greatly enhance data handling and querying capabilities.

Here's the official Microsoft reference for all T-SQL string functions:

**Microsoft Documentation:**

- String Functions (Transact-SQL):https://learn.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-ver16

This comprehensive resource provides detailed descriptions, syntax, and examples for a wide range of string functions, including:

- Basic string manipulation (LEN, UPPER, LOWER, SUBSTRING, CHARINDEX, REPLACE, etc.)
- Formatting and padding (FORMAT, REPLICATE, SPACE, STR)
- Pattern matching (PATINDEX, LIKE)
- String comparison (DIFFERENCE, SOUNDEX)
- String aggregation (STRING_AGG, STRING_ESCAPE, STRING_SPLIT)
- Truncation and padding (STUFF, TRIM)
- Unicode string handling (UNICODE)

I highly recommend exploring this documentation to gain a deeper understanding of the available string functions and their capabilities.

## Date Functions

دهبعض ال functions الخاصه بالتواريخ

getdate بترجعلك تاريخ النهارده

systemdate بيرجعلك تاريخ النهارده بس بدقه اعلي

date add بتضيف أيام او اشهر او سنين علي تاريخ معين ولو عايز تطرح بتستخدم الرقم بالسالب

date diff بتجيبلك الفرق بين تاريخين شواء بالايام او الشهور او السنين

date part بتجيبلك جزء من التاريخ

datename بيجيبلك اسم الشهر او اليوم من تاريخ معين

day بيجيبلك اليوم من التاريخ كرقم

month بيجيبلك الشهر من تاريخ معين

year بيجيبلك السنه من تاريخ معين

convert بتقدر تحول التاريخ لنص وال 103 بتمثل ال format بتاع التاريخ

cast بتحول التاريخ معاه وقت لتاريخ بس من غير وقت

EOmonth بترجعلك اخر يوم في الشهر

```
use C21_DB1;

-- Common Date funcitons:

-- Getting the current system date and time
SELECT GETDATE() AS CurrentDateTime
GO

-- Getting the system date and time with fractional seconds and time zone offset
SELECT SYSDATETIME() AS SystemDateTime
GO

-- Adding 10 days to the current date
SELECT DATEADD(day, 10, GETDATE()) AS DatePlus10Days
GO

-- Calculating the difference in days between two dates
SELECT DATEDIFF(day, '2023-01-01', GETDATE()) AS DaysSinceStartOfYear
GO

-- Extracting the year part from the current date
SELECT DATEPART(year, GETDATE()) AS CurrentYear
GO

-- Getting the name of the current month
SELECT DATENAME(month, GETDATE()) AS CurrentMonthName
GO

-- Extracting the day from the current date
SELECT DAY(GETDATE()) AS CurrentDay
GO

-- Extracting the month from the current date
SELECT MONTH(GETDATE()) AS CurrentMonth
GO
```

```
-- Extracting the year from the current date
SELECT YEAR(GETDATE()) AS CurrentYear
GO

-- Converting a datetime to a different format,The third argument, 103, specifies the style code for
the conversion.
--In SQL Server, style code 103 represents the date format as DD/MM/YYYY.
--This means that the resulting string will have the day, then the month, and finally the year,
separated by forward slashes.
SELECT CONVERT(varchar, GETDATE(), 103) AS DateInDDMMYYYY
GO

-- Casting a datetime to a different data type
SELECT CAST(GETDATE() AS date) AS DateOnly
GO

-- Getting the last day of the current month
SELECT EOMONTH(GETDATE()) AS LastDayOfCurrentMonth
GO
```

## Overview of Date Functions in T-SQL

Date functions in T-SQL are crucial for manipulating and querying datetime data. These functions allow you to format dates, calculate time intervals, and extract specific parts of a date. Here's an overview of some commonly used date functions in T-SQL:

1. GETDATE()

- Purpose: Returns the current date and time.

2. DATEADD()

- Purpose: Adds a specified number of units (days, months, years, etc.) to a date.

3. DATEDIFF()

- Purpose: Calculates the difference between two dates in a specified unit (days, months, years, etc.).

4. DATEPART()

- Purpose: Returns a specified part of a date, such as year, month, day, etc.

5. DATENAME()

- Purpose: Returns the name of the specified part of a date, such as the name of the month, day of the week, etc.

6. DAY()

- Purpose: Extracts the day part of a date.

7. MONTH()

- Purpose: Extracts the month part of a date.

8. YEAR()

- Purpose: Extracts the year part of a date.

9. CONVERT()

- Purpose: Converts an expression of one data type to another, often used to format dates.

10. CAST ()

- Purpose: Similar to CONVERT, it changes the data type of an expression, frequently used in date formatting.

11. EOMONTH()

- Purpose: Returns the last day of the month for a specified date.

12. SYSDATETIME()

- Purpose: Returns the system date and time with fractional seconds and time zone offset.

These functions are integral in managing and analyzing date and time data in SQL Server. Effective use of these functions can significantly enhance your data querying and manipulation capabilities.

Here's the official Microsoft reference for all T-SQL date functions:

**Microsoft Documentation**:

**Date and Time Data Types and Functions (Transact-SQL):**

https://learn.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql?view=sql-server-ver16

This comprehensive resource provides detailed descriptions, syntax, and examples for a wide range of date and time functions, including:

**Functions that return system date and time values:**

- GETDATE()
- SYSDATETIME()
- SYSUTCDATETIME()
- CURRENT_TIMESTAMP

**Functions that return date and time parts:**

- DATEPART()
- DATENAME()
- YEAR()
- MONTH()
- DAY()
- DATEDIFF()
- EOMONTH()

**Functions that return date and time values from their parts:**

- DATEFROMPARTS()
- SMALLDATETIMEFROMPARTS()
- DATETIME2FROMPARTS()
- DATETIMEOFFSETFROMPARTS()

**Functions that modify date and time values:**

- DATEADD()
- DATEDIFF()

**Functions that set or return session format functions:**

- FORMAT()

**Functions that validate date and time values:**

- ISDATE()

**Date and time-related articles:**

- CAST and CONVERT (Transact-SQL)
- SWITCHOFFSET (Transact-SQL)
- TODATETIMEOFFSET (Transact-SQL)

I highly recommend exploring this documentation to gain a deeper understanding of the available date functions and their capabilities.

## Aggregate functions (SUM, AVG, COUNT, MIN, MAX)

دي مراجعه علي ال aggregate functions

In T-SQL, there are several aggregate functions that allow you to perform calculations on sets of data, typically used in combination with the `GROUP BY` clause to summarize data within groups. Let's explore each aggregate function with an example using the "Employees2" table you've created.

**Aggregate Functions:**

**COUNT()** - Counts the number of rows in a group or result set.

```
-- Example: Count the number of employees in each department
SELECT Department, COUNT(*) AS EmployeeCount
FROM Employees2
GROUP BY Department;
```

**SUM()** - Calculates the sum of values in a numeric column.

```
-- Example: Calculate the total salary for each department
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees2
GROUP BY Department;
```

**AVG()** - Calculates the average value of a numeric column.

```
-- Example: Calculate the average performance rating for each department
SELECT Department, AVG(PerformanceRating) AS AvgPerformanceRating
FROM Employees2
GROUP BY Department;
```

**MIN()** - Retrieves the minimum value in a column.

```
-- Example: Find the lowest salary in the company
SELECT MIN(Salary) AS LowestSalary
FROM Employees2;
```

**MAX()** - Retrieves the maximum value in a column.

```
-- Example: Find the highest salary in the company
SELECT MAX(Salary) AS HighestSalary
FROM Employees2;
```

**Examples:**

COUNT(): To count the number of employees in each department:

```
SELECT Department, COUNT(*) AS EmployeeCount
FROM Employees2
GROUP BY Department;
```

**SUM():** To calculate the total salary for each department:

```
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees2
GROUP BY Department;
```

**AVG():** To calculate the average performance rating for each department:

```
SELECT Department, AVG(PerformanceRating) AS AvgPerformanceRating
FROM Employees2
GROUP BY Department;
```

**MIN():** To find the lowest salary in the company:

```
SELECT MIN(Salary) AS LowestSalary
FROM Employees2;
```

**MAX():** To find the highest salary in the company:

```
SELECT MAX(Salary) AS HighestSalary
FROM Employees2;
```

These are just a few examples of how you can use aggregate functions in T-SQL to summarize and analyze data in your tables. Aggregate functions are powerful tools for data analysis and reporting in SQL.

## What is Window Functions?

ال aggregate functions يعتبروا window functions

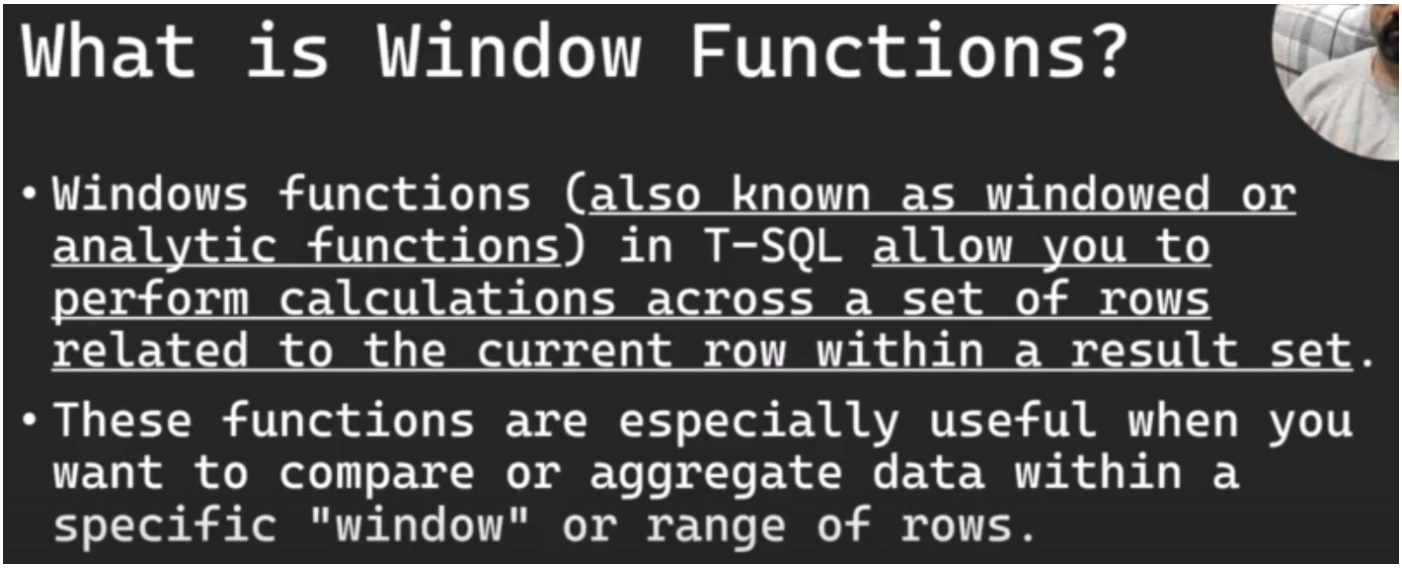
ال window functions هيا functions بتخليك تعمل calculations علي dataset معينه وتعمل مقارنه بينهم وتجيب statistical information ليهم ومهم انك تعرفهم عشان بيوفروا كود كبير عليك

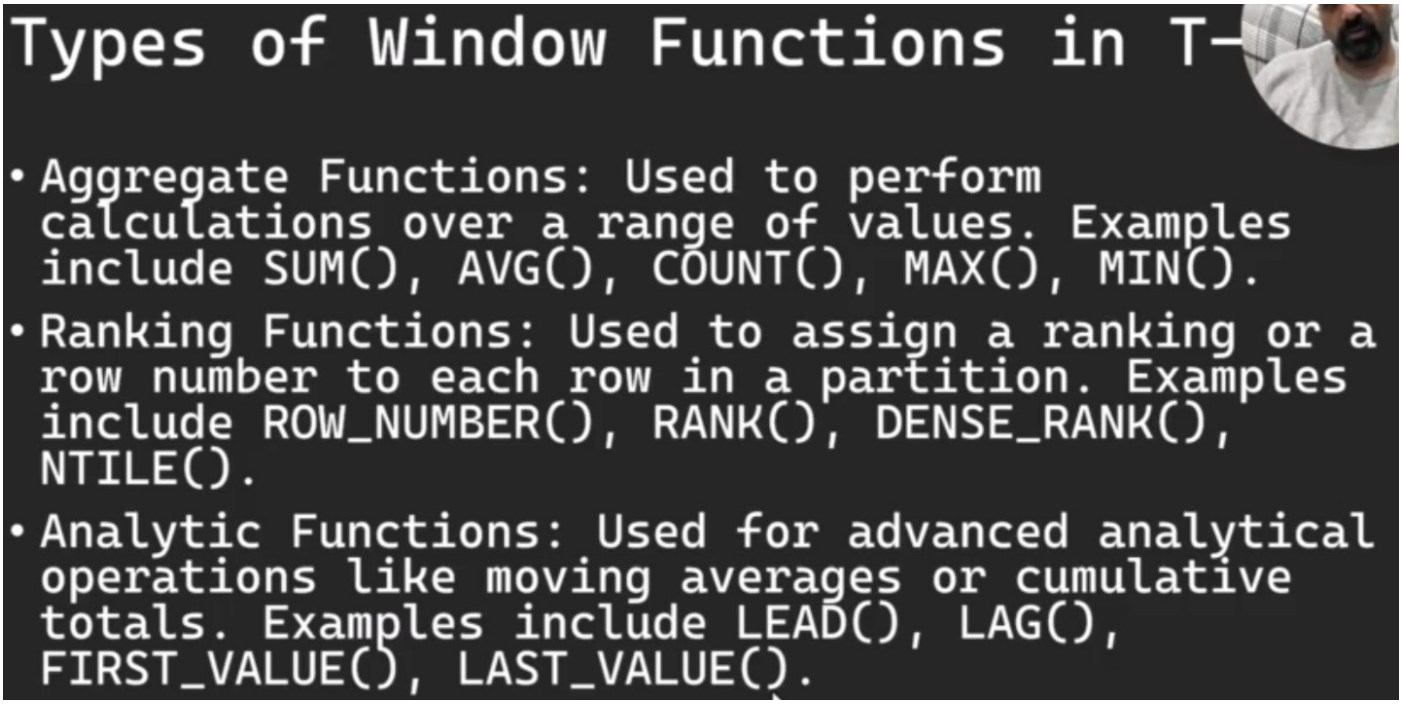
فيه منهم أنواع

aggregate functions زي الsum وال avg

ranking functions هناخد عليهم امثله

analytical functions






**What is Window Functions:**

Windows functions (also known as windowed or analytic functions) in T-SQL allow you to perform calculations across a set of rows related to the current row within a result set. These functions are especially useful when you want to compare or aggregate data within a specific "window" or range of rows.

**Types of Window Functions in T-SQL**

- Aggregate Functions: Used to perform calculations over a range of values. Examples include `SUM()`, `AVG()`, `COUNT()`, `MAX()`, `MIN()`.
- Ranking Functions: Used to assign a ranking or a row number to each row in a partition. Examples include `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, `NTILE()`.
- Analytic Functions: Used for advanced analytical operations like moving averages or cumulative totals. Examples include `LEAD()`, `LAG()`, `FIRST_VALUE()`, `LAST_VALUE()`.

## Understanding the ROW_NUMBER() Function in SQL

ROW_NUMBER دي FUNCTION وظيفتها انك لو بترجع داتا معينه هوا بيزود عمود فيه مسلسل او ترقيم للصفوف اللي خارجه

اعمل الجدول ده الأول

```
use C21_DB1;
-- Create the Students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(50),
```

```
    Subject NVARCHAR(50),
    Grade INT
);


Go

-- Insert sample data into the Students table
INSERT INTO Students (StudentID, Name, Subject, Grade)
VALUES
    (1, 'Alice', 'Math', 90),
    (2, 'Bob', 'Math', 85),
    (3, 'Charlie', 'Math', 78),
    (4, 'Dave', 'Science', 88),
    (5, 'Emma', 'Science', 92),
    (6, 'Fiona', 'Science', 84),
    (7, 'Grace', 'English', 75),
    (8, 'Henry', 'English', 80),
    (9, 'Isabella', 'English', 88);
```

طيب دلوقتي هاتلي كل الداتا اللي في الجدول بس رتبهم تنازليا حسب الدرجات

```
use C21_DB1;

select * from Students order by Grade desc
```

طيب دلوقتي لو قولتلك رقمهم بعد الترتيب اللي عملته ؟

هنا هنضيف عمود يرتبهم

```
use C21_DB1
select Students.StudentID,
Students.Name,
Students.Subject,
Students.Grade,
RowNum=ROW_NUMBER() over (order by Students.Grade desc)
from Students
```

# Understanding the ROW_NUMBER() Function in SQL

**Introduction**

The `ROW_NUMBER()` function in SQL is a window function that assigns a unique sequential integer to rows within a partition of a result set, starting at 1.

Unlike `RANK()` and `DENSE_RANK()`, `ROW_NUMBER()` does not assign the same number to ties. It's ideal for scenarios where you need a distinct identifier for each row, regardless of duplicates in the ordering column.

**Scenario**

We will use the `Students` table, which includes `StudentID`, `Name`, `Subject`, and `Grade`, to demonstrate how `ROW_NUMBER()` can be used to assign a unique number to each student based on their grade.

**SQL Concepts: ROW_NUMBER()**

- 
  - ROW_NUMBER():Assigns a unique sequential number to each row.
  - Starts at 1 and increases by 1 for each row.
  - If there are ties (e.g., two students with the same grade), each row still gets a unique number.

**Using ROW_NUMBER()**

Example Query:

```
SELECT
    StudentID,
    Name,
    Subject,
    Grade,
    ROW_NUMBER() OVER (ORDER BY Grade DESC) AS RowNum
FROM
    Students;
```

- This query assigns a unique row number to each student, ordered by their grade in descending order.
- The `RowNum` column will show this unique number.

**Understanding the Output**

- Each student will have a unique `RowNum`, even if two or more students have the same grade.
- The student with the highest grade gets `RowNum` = 1, the next gets `RowNum` = 2, and so on, regardless of ties.

**Conclusion**

`ROW_NUMBER()` is particularly useful for creating a unique identifier for each row in a result set, which can be beneficial for pagination or when processing data in ordered chunks.

Through this lesson, you'll understand how to use the `ROW_NUMBER()` function in SQL for assigning distinct sequential numbers to rows within a dataset, a crucial technique in data analysis and manipulation.

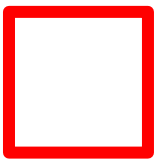
## Understanding the RANK() Function in SQL

ال rank زيها زي ال row number بس الفرق بينهم انه ال rank لو شاف قيمتين زي بعض بيديهم نفس الرقم

زي الأول والأول مكرر كده

```
use C21_DB1
select Students.StudentID,
Students.Name,
Students.Subject,
Students.Grade,
RankNum=Rank() over (order by Students.Grade desc)
from Students
```

بس فيه مشكله انه لما بيكرر رقم بيلغي الرقم اللي بعده زي كده هنحلها في الدرس الجاي



Results Messages

| | StudentID | Name | Subject | Grade | RankNum |
|---|---|---|---|---|---|
| 1 | 5 | Emma | Science | 92 | 1 |
| 2 | 1 | Alice | Math | 90 | 2 |
| 3 | 4 | Dave | Science | 88 | 3 |
| 4 | 9 | Isabella | English | 88 | 3 |
| 5 | 2 | Bob | Math | 85 | 5 |
| 6 | 6 | Fiona | Science | 84 | 6 |
| 7 | 8 | Henry | English | 80 | 7 |
| 8 | 3 | Charlie | Math | 78 | 8 |
| 9 | 7 | Grace | English | 75 | 9 |

## Understanding the RANK() Function in SQL

**Introduction**

In SQL, ranking functions are used to provide sequential numbering of the rows in the result set. The `RANK()` function is one of these functions, and it assigns a rank to each row within a partition of a result set.

**Scenario**

We have a `Students` table with columns `StudentID`, `Name`, `Subject`, and `Grade`. We will use the `RANK()` function to assign a rank to each student based on their grade.

SQL Concept: RANK() Function

- RANK(): This function assigns a rank to each row within a partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.

**Using RANK()**

Let's write a query to rank students based on their grades:

```
SELECT
    StudentID,
    Name,
    Subject,
    Grade,
    RANK() OVER (ORDER BY Grade DESC) AS GradeRank
FROM
    Students;
```

In this query:

- We use the `RANK()` function within a `SELECT` statement.
- `OVER (ORDER BY Grade DESC)` determines the order of the ranking. Here, students are ranked based on their grades, in descending order (highest grade gets rank 1).
- `GradeRank` is an alias for the new column that will display the rank of each student.

Understanding the Output

- Students with the highest grade will be ranked 1.
- If two or more students share the same grade, they will receive the same rank. The next rank will be incremented by the total number of students with the previous grade.

Conclusion

The `RANK()` function is useful for ranking rows in a dataset. In our case, it helps in understanding the relative performance of students based on their grades.

Exercise

1. Insert some sample data into the `Students` table and run the ranking query.
2. Experiment with ranking based on different columns (e.g., `Subject`).
3. Try changing the order (ASC, DESC) in the `OVER` clause to see how it affects the ranking.

This lesson provides a practical understanding of how to use the `RANK()` function in SQL, which is essential for tasks involving sorting and ranking data within databases.

## Understanding the Difference Between RANK and DENSE_RANK in SQL

عشان نحل مشكله الأرقام اللي بتطير في ال rank بنستخدم function اسمها dense_rank

```
use C21_DB1
select Students.StudentID,
Students.Name,
Students.Subject,
Students.Grade,
RankNum=DENSE_Rank() over (order by Students.Grade desc)
from Students
```

## Understanding the Difference Between RANK and DENSE_RANK in SQL

**Introduction**

In SQL, both `RANK()` and `DENSE_RANK()` are window functions used to assign ranks to rows in a dataset. Although similar, they have distinct ways of handling ties (rows with equal values in the ordered column). Understanding the difference is crucial for effectively applying these functions in data analysis.

**Scenario**

Imagine we have a `Students` table with columns for `StudentID`, `Name`, `Subject`, and `Grade`. We want to rank students based on their grades.

**SQL Concepts: RANK vs. DENSE_RANK**

- **RANK():**
    - Assigns a unique rank to each row within a result set.
    - Ranks are assigned in the order specified (e.g., descending grades).
    - If two or more rows tie (same grade), they receive the same rank.
    - The next rank after a tie is incremented by the total number of tied rows. For example, if two students are tied for rank 1, the next student will receive rank 3.
- **DENSE_RANK():**
    - Similar to `RANK()`, assigns ranks within a result set.
    - Handles ties like `RANK()` but does not skip ranks after ties.
    - If there are ties, the next rank after a tie is incremented by one. For example, if two students are tied for rank 1, the next student will receive rank 2.

### Example Query

Using `RANK()`:

```
SELECT
    StudentID,
    Name,
    Grade,
    RANK() OVER (ORDER BY Grade DESC) AS GradeRank
FROM
    Students;
```

Using `DENSE_RANK()`:

```
SELECT
    StudentID,
    Name,
    Grade,
    DENSE_RANK() OVER (ORDER BY Grade DESC) AS GradeRank
FROM
```

```
    Students;
```

**Understanding the Output**

Consider this set of grades: [95, 95, 90, 85, 85, 85, 80]

- Using `RANK()`, the ranks would be [1, 1, 3, 4, 4, 4, 7].
- Using `DENSE_RANK()`, the ranks would be [1, 1, 2, 3, 3, 3, 4].

**Conclusion**

Choose `RANK()` when you need to account for gaps in ranking after ties. Use `DENSE_RANK()` when you want a continuous ranking sequence without gaps.

Exercise

1. Insert sample data into the `Students` table with some tied grades.
2. Run both queries and compare the results.
3. Experiment with different orderings and partitioning to see how it affects the ranks.

Through this lesson, you gain an understanding of how to use `RANK()` and `DENSE_RANK()` in SQL and their distinct approaches to handling ties in data ranking. This knowledge is essential for data analysis tasks where ranking is involved.

## Using the RANK() Function with PARTITION BY in SQL

ال PARTITION BY دي بتجزء الداتا زي ال GROUP BY كده

لو استخدمناها مع ال rank هترقم الطلبه حسب الماده

يعني مين الأول في ماده الرياضيات ومين الأول في ماده التاريخ وهكذا

```
use C21_DB1
select Students.StudentID,
Students.Name,
Students.Subject,
Students.Grade,
RankNum=Rank() over (partition by Subject order by
Students.Grade desc)
from Students
```

وممکن کده برضه

```
use C21_DB1
select Students.StudentID,
Students.Name,
Students.Subject,
Students.Grade,
RankNum=DENSE_Rank() over (partition by Subject order by
Students.Grade desc)
from Students
```

## Using the RANK() Function with PARTITION BY in SQL

**Introduction**

In SQL, the `PARTITION BY` clause is used in conjunction with window functions like `RANK()`. This clause allows you to divide the result set into partitions and apply the ranking function within each partition.

Scenario

Continuing with our `Students` table, we will now rank students within each subject. This means that the rank will restart for each subject.

**SQL Concepts: PARTITION BY Clause**

- PARTITION BY: This clause divides the result set into partitions where the ranking function restarts its count for each partition.

**Using RANK() with PARTITION BY**

Let's modify our previous query to include `PARTITION BY`:

```
SELECT
    StudentID,
    Name,
    Subject,
    Grade,
    RANK() OVER (PARTITION BY Subject ORDER BY Grade DESC) AS GradeRank
FROM
    Students;
```

In this query:

- `PARTITION BY Subject` means the ranking will be reset for each subject.
- `ORDER BY Grade DESC` still orders the students by grade within each subject.
- `GradeRank` shows the rank of students within each specific subject.

Understanding the Output

- Students are ranked within each subject based on their grades.
- If students in the same subject have the same grade, they will have the same rank. The next rank is incremented based on the total number of students with the previous grade within that subject.
- The rank resets for each different subject.

Conclusion

Using `RANK()` with `PARTITION BY` allows for more nuanced analysis of data, enabling ranking within specific subsets of data. This is particularly useful in scenarios where comparative ranking is required within categories.

Exercise

1. Insert sample data into the `Students` table, ensuring multiple subjects are represented.
2. Run the modified query and observe how students are ranked within each subject.
3. Experiment by changing the partitioning column to something else, like `Grade`.

This lesson highlights the utility of partitioning in SQL queries, specifically for tasks that require categorized analysis or sorting within specific groups. The `PARTITION BY` clause, when used with functions like `RANK()`, offers a powerful tool for sophisticated data manipulation and analysis in SQL.

## Aggregate Functions with Partition

لو قولتلك هاتلي الجدول بتاع الدرجات وقدام كل درجه حطي مجموع الدرجات والمتوسط بتاعها

```
use C21_DB1
select Students.StudentID,
Students.Name,
```

```
Students.Subject,
Students.Grade,
SumSubject=Sum(Students.Grade) over (partition by
Students.Subject),
AVGSubject=AVG(Students.Grade) over (partition by
Students.Subject)
from Students
```

## Aggregate Functions with Partition

**Introduction**

Window functions in SQL provide a way to perform calculations across sets of rows related to the current row. This lesson focuses on the `AVG()` and `SUM()` window functions used with the `PARTITION BY` clause. We will explore how these functions can calculate average and total values within each partition of the data.

**Scenario**

Using the `Students` table, which includes columns `StudentID`, `Name`, `Subject`, and `Grade`, we will demonstrate how to calculate the average and total grade for each subject.

**SQL Concepts: AVG and SUM with PARTITION BY**

- AVG():
  - Computes the average of a set of values.
  - In combination with `OVER (PARTITION BY)`, it calculates the average within each partition.
- SUM():
  - Calculates the total sum of a numeric column.
  - Used with `OVER (PARTITION BY)`, it sums up values within each partition.

**Using AVG and SUM with PARTITION BY**

**Example Query:**

```
SELECT
    StudentID,
```

```
    Name,
    Subject,
    Grade,
    AVG(Grade) OVER (PARTITION BY Subject) AS SubjectAvgGrade,
    SUM(Grade) OVER (PARTITION BY Subject) AS SubjectTotalGrade
FROM
    Students
ORDER BY Subject;
```

- This query calculates the average (`SubjectAvgGrade`) and total (`SubjectTotalGrade`) grade for each subject.
- `PARTITION BY Subject` groups the data by subject, so each subject is considered a separate partition for the calculations.

**Understanding the Output**

- `SubjectAvgGrade` shows the average grade for each subject.
- `SubjectTotalGrade` shows the total sum of grades for each subject.
- Each row will display these calculated values alongside individual student data.
- The result is ordered by `Subject`, grouping students by their subject.

**Conclusion**

Using `AVG()` and `SUM()` with `PARTITION BY` in window functions allows for complex calculations within specific subsets of data. These are essential for analyzing grouped data, like calculating averages and totals within categories.

This lesson illustrates the power and flexibility of window functions in SQL, particularly useful for data analysis tasks requiring aggregated calculations within distinct partitions of a dataset.

## Exploring LAG and LEAD Functions Using a Single SQL Query

لو قولتلك هاتلي جدول ال students ورتبه تنازليا حسب ال grade تعمل ايه؟

هتقولي سهله خد اهو

```
use C21_DB1
select * from students order by Students.Grade desc
```

طيب لو قولتلك هاتلي عمود ال name وال id والgrade بعد ماترتبهم تنازليا من حيث الgrade وزود عليهم عمودين من عندك

اول عمود يمسك كل record ويبص عال grade اللي فيه ويجيب ال grade بتاعت ال record اللي قبله بعد مارتبتهم

وتاني عمود يجيب ال grade اللي بعده

اللي هيخليك تعمل ده بسهوله functions 2

واحده اسمها LAG ودي اللي هتخليك تجيب القيمه اللي في ال record اللي قبله

والتانيه اسمها lead ودي اللي هتخليك تجيب القيمه اللي في ال record اللي بعده

```
select Students.StudentID,Students.Name,
PreviousGrade = LAG(Students.Grade,1) over (order by
Students.Grade desc),
Students.Grade,
NextGrade = LEAD(Students.Grade,1) over (order by
Students.Grade desc)
from Students
```

## Exploring LAG and LEAD Functions Using a Single SQL Query

**Introduction**

In SQL, the `LAG` and `LEAD` functions are invaluable for comparing data across rows without complex joins. These window functions fetch data from preceding or following rows in the same result set. We will use a single query on the `Students` table to illustrate how both functions can be simultaneously applied.

Scenario

Using the `Students` table, we will compare each student's grade with the grades of the preceding and following students.

**SQL Concepts: LAG and LEAD Together**

- 
  - LAG():Retrieves data from a preceding row in the result set.
  - Used for comparing the current row with past data.
  - LEAD():Retrieves data from a following row in the result set.

- Used for comparing the current row with future data.

### Using LAG and LEAD in One Query

```
SELECT
    StudentID,
    Name,
    LAG(Grade, 1) OVER (ORDER BY Grade DESC) AS PreviousGrade,
    Grade,
    LEAD(Grade, 1) OVER (ORDER BY Grade DESC) AS NextGrade
FROM
    Students
ORDER BY Grade DESC;
```

### In this query:

- We use `LAG` to fetch the grade of the student who has the next lower grade (PreviousGrade).
- We use `LEAD` to fetch the grade of the student who has the next higher grade (NextGrade).
- Both functions are applied over the dataset ordered by `Grade DESC`.

### Understanding the Output

- The `PreviousGrade` column shows the grade of the student ranked immediately lower.
- The `NextGrade` column shows the grade of the student ranked immediately higher.
- The dataset is ordered by `Grade DESC`, so the comparison is in the context of descending grades.

### Conclusion

Combining `LAG` and `LEAD` in a single query allows for a comprehensive comparison within the dataset, revealing patterns and relationships that might not be immediately apparent.

This lesson demonstrates the combined power of `LAG` and `LEAD` in SQL for analyzing ordered datasets, especially useful in scenarios requiring sequential data comparison.

## Paging in SQL using OFFSET and FETCH NEXT

الفكره هنا انك لو بترجع داتا حجمها كبير جدا ده بيزود ال traffic وكمان ممكن اللي اليوزر بيدور عليه يلاقيه في اول الجدول اللي راجع ده

زي مثلا انك تفتح صفحة ويب او يوتيوب وتعمل بحث عن حاجه معينه

الصفحه ممكن تسحب باقة النت كلها وتبقي تقيله وتاخد وقت كبير عبال ماتحمل

فقالك ليه كل ده؟ ماتقسم الداتا اللي راجعه لليوزر بحيث تحط كل شوية منهم في صفحه او تحمل جزء منهم ولما يبقي يطلب تاني ابقي ابعتله

المفهوم ده بيسموه paging انك بتختار عدد معين من ال records ولما اليوزر يطلب تاني ترجعله الجزء اللي بعده كانك بتقسمهم لصفحات

طيب هتعمل ده ازاي؟

قالك فيه اتنين functions

واحده بتخليك تعمل skip لعدد معين من ال records وترجعلك الباقي واسمها offset

والتانيه بتعمل زي select لعدد معين من ال records من بعد ال offset واسمها fetch next

وبتستخدمهم مع بعض بعد ماتعمل ال query بتاعتك ولازم يكون فيها order by

تعالي نعملها واحده واحده مع بعض

اول حاجه تعالي نعمل 2variables واحد بيمثل رقم الصفحه والتاني بيمثل عدد الrecords في الصفحه الواحده

```
declare @PageNumber int=2
declare @RowsPerPage int=3
```

بعدين هنعمل ال query بتاعتنا عادي

```
select * from Students order by Students.StudentID
```

بعدين هنستخدم ال offset وال fetch next

ال offset بتاخد رقم بيمثل عدد ال records اللي عايز اعملها skip

وال fetch next بحددله عدد الصفوف اللي عايز ارجعها

```
offset (@PageNumber-1)*@RowsPerPage rows
fetch next @RowsPerPage rows only
```

وده الكود كله

```
use C21_DB1
```

```
declare @PageNumber int=2
declare @RowsPerPage int=3

select * from Students order by Students.StudentID
offset (@PageNumber-1)*@RowsPerPage rows
fetch next @RowsPerPage rows only
```

## Paging in SQL using OFFSET and FETCH NEXT

**Introduction**

Paging is a critical feature in database management, particularly useful when dealing with large datasets. It allows for displaying data in segmented chunks, enhancing performance and user experience. SQL Server offers the `OFFSET` and `FETCH NEXT` clauses to implement paging efficiently.

**Scenario**

We have a `Students` table with multiple records. Our objective is to display this data in a paginated format. In this example, each page will show 3 students, and we'll focus on retrieving the second page of results.

SQL Concepts: OFFSET and FETCH NEXT

- OFFSET: Skips a set number of rows in the data.
- FETCH NEXT: Retrieves a specific number of rows after the offset.

**Setting Up Variables**

We use variables for dynamic paging control:

```
DECLARE @PageNumber AS INT, @RowsPerPage AS INT;
SET @PageNumber = 2;  -- Set to the second page
SET @RowsPerPage = 3; -- Displaying 3 rows per page
```

`@PageNumber` is now set to 2, indicating we want the second page of data. `@RowsPerPage` remains at 3.

**Implementing Paging**

To fetch the data for the second page:

```
SELECT StudentID, Name, Subject, Grade
FROM Students
ORDER BY StudentID
OFFSET (@PageNumber - 1) * @RowsPerPage ROWS
FETCH NEXT @RowsPerPage ROWS ONLY;
```

- We use `ORDER BY StudentID` for consistent ordering.
- `OFFSET (@PageNumber - 1) * @RowsPerPage ROWS`: This now skips the first 3 rows (since `@PageNumber` is 2, so `(2-1)*3 = 3`).
- `FETCH NEXT @RowsPerPage ROWS ONLY`: Retrieves the next 3 rows after the offset, which constitutes the second page of data.

**Example**

- Page 1: Shows the first 3 students.
- Page 2 (current): Skips the first 3 students, showing students 4 to 6.
- Page 3: Would skip the first 6 students, showing students 7 to 9, and so on.

**Conclusion**

By adjusting `@PageNumber`, you can navigate through different pages in a dataset. `OFFSET` and `FETCH NEXT` provide a straightforward method to implement paging in SQL Server, which is essential for managing and displaying large volumes of data efficiently.

Exercises

1. Change `@PageNumber` to access different pages (e.g., 1, 3, 4).
2. Experiment with `@RowsPerPage` to display different numbers of rows per page.
3. Try different `ORDER BY` clauses to understand their impact on the data presentation.

This lesson emphasizes the flexibility and importance of paging in database systems, particularly for improving data manageability and user interface design.

**Quiz**

What are window functions in T-SQL?

- Functions that allow you to perform calculations across a set of rows related to the current row within a result set.
- Functions that allow you to perform calculations on aggregate values.
- Functions that assign a ranking or a row number to each row in a partition.
- Functions that perform advanced analytical operations.

Which type of window function is used to perform calculations over a range of values?

- Aggregate Functions
- Ranking Functions
- Analytic Functions
- None of the above

Which type of window function is used to assign a ranking or a row number to each row in a partition?

- Aggregate Functions
- Ranking Functions
- Analytic Functions
- None of the above

Which type of window function is used for advanced analytical operations like moving averages or cumulative totals?

- Aggregate Functions
- Ranking Functions
- Analytic Functions
- None of the above

## Scalar Functions in T-SQL

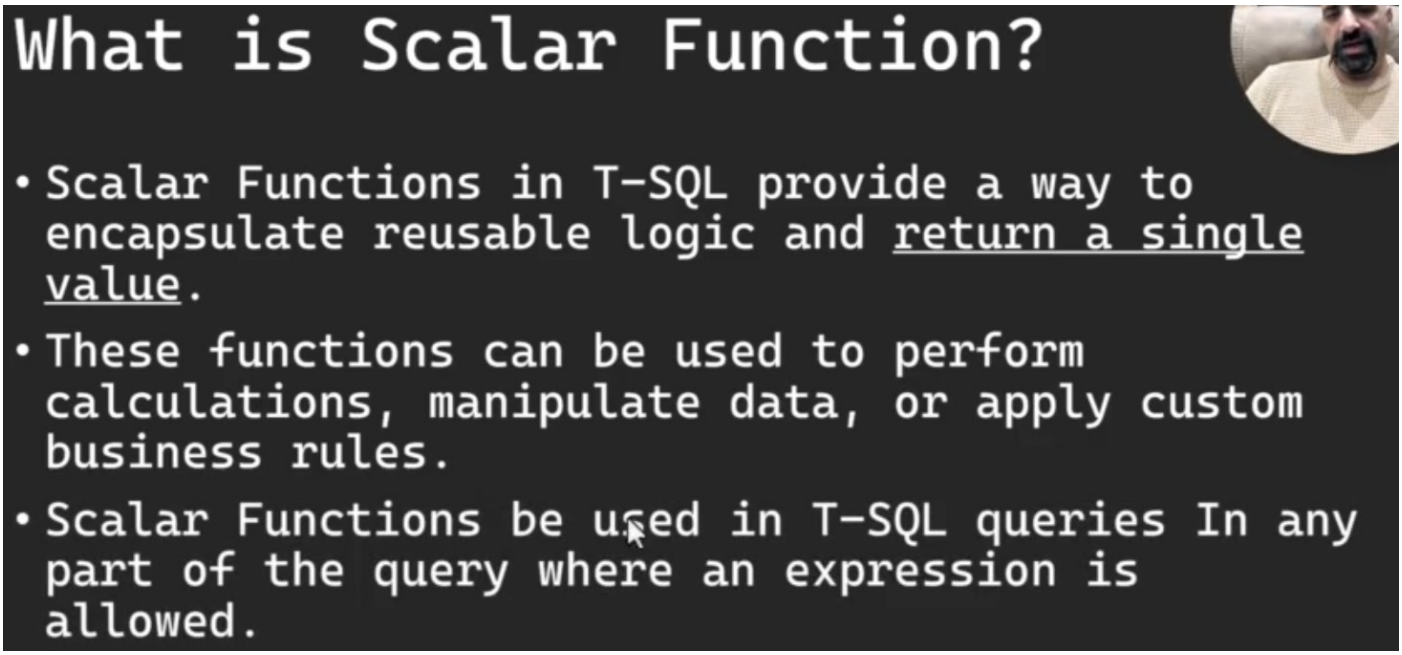
كنا قبل كده اتكلمنا عن ال stored procedures وقولنا انها عبارة عن functions بس ماتقدرش تستخدمها جوه ال query

هنا بقي هنعرف ازاي نعمل function نقدر نستخدمها جوه ال query

ال functions دي بتكون read only يعني تقدر تحط جواها select statement بس ماتقدرش تحط update او insert

ال functions فيه منها أنواع

منها ال scalar function ودي function مش بترجع غير قيمه واحده بس




تعالي نضيف الجدول ده

```
use C21_DB1
CREATE TABLE Teachers (
    TeacherID INT PRIMARY KEY,
```

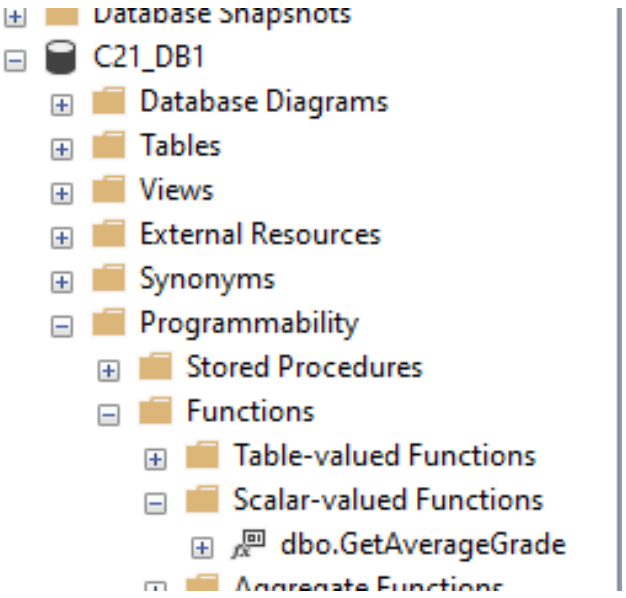
```
    Name NVARCHAR(50),
    Subject NVARCHAR(50)
);


INSERT INTO Teachers (TeacherID, Name, Subject) VALUES
(1, 'John Smith', 'Math'),
(2, 'Jane Doe', 'Science'),
(3, 'Emily Johnson', 'English'),
(4, 'Mark Brown', 'History'),
(5, 'Sarah Davis', 'Music');
```

طيب تعالي نعمل function تاخد اسم الماده وتروح تحسبلنا المتوسط بتاعها

عشان نعمل function بنكتب create function وبعدين نديها اسم ونفتح قوسين نكتب ال parameters وبعد القوسين بنكتب return ونحدد ال return type وبعدين نكتب الكود بتاعنا ونرجع القيمه زيها زي أي function بنعملها

```
create function GetAverageGrade (@Subject nvarchar(50))
returns int

as
begin
declare @AverageGrade int

select @AverageGrade=avg(Students.Grade)
from Students
where Students.Subject=@Subject

return @AverageGrade
end
```

لما بتشغل الكود بتلاقي ال function هنا



طيب تعالي بقي هاتلي من جدول ال teachers اسم كل مدرس وقدامه متوسط الدرجات بتاع الماده اللي بيدرسها

```
select Teachers.Name,
AverageGrade=dbo.GetAverageGrade(Teachers.Subject)
from Teachers
```

طيب اعملي function تانيه تاخد performance rate و salary وترجع bonus ولو كان ال rate اعلي من او يساوي 5 ال bonus هيكون 10% من الراتب ولوغير كده احسبله 5%

```
create function CalculateBonus(@PerformanceRate int,@Salary
int)
returns int

as
begin
declare @Bonus int

if @PerformanceRate>=5

set @Bonus =@Salary*.1
else
set @Bonus=@Salary*.05

return @Bonus
end
```

تعالي نجربها

```
select Bonus=dbo.CalculateBonus(5,1000)
```

**Scalar Functions in T-SQL**

Understand how to create and use Scalar Functions in T-SQL to encapsulate reusable logic and return a single value.

**Introduction:**

Scalar Functions in T-SQL provide a way to encapsulate reusable logic and return a single value. These functions can be used to perform calculations, manipulate data, or apply custom business

rules. This lesson will guide you through the process of creating and using Scalar Functions based on the previously provided table scripts.

Scalar Functions be used in T-SQL queries In any part of the query where an expression is allowed.

**Recap of Previously Created Objects:**
Before we proceed, let's recap the previously created objects:

**Table Scripts:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT
);


CREATE TABLE Teachers (
    TeacherID INT PRIMARY KEY,
    Name NVARCHAR(50),
    Subject NVARCHAR(50)
);
```

**Explanation:**

- We have two tables: Students and Teachers, which store information about students and teachers, respectively.

**Creating a Scalar Function:**
Now, let's create a Scalar Function named "GetAverageGrade".

```
CREATE FUNCTION dbo.GetAverageGrade(@subject NVARCHAR(50))
RETURNS INT
AS
BEGIN
    DECLARE @averageGrade INT;
```

```
    SELECT @averageGrade = AVG(Grade)
    FROM Students
    WHERE Subject = @subject;

    RETURN @averageGrade;
END;
```

**Explanation:**

- In the above script, we create a Scalar Function named "GetAverageGrade".
- The function takes a parameter @subject of type NVARCHAR(50) to specify the subject for which we want to calculate the average grade.
- Inside the function, we declare a variable @averageGrade to hold the calculated average grade.
- We use the SELECT statement to calculate the average grade from the Students table, filtered by the specified subject.
- Finally, we return the @averageGrade as the result of the function.

**Using the Scalar Function:**

Now that we have created the Scalar Function, let's use it in a query.

```
SELECT Name, dbo.GetAverageGrade(Subject) AS AverageGrade
FROM Teachers;
```

**Explanation:**

- We use the SELECT statement to retrieve data from the Teachers table.
- We also call the Scalar Function dbo.GetAverageGrade(Subject) to calculate the average grade for each teacher's subject.

- The result set will contain the names of the teachers and their corresponding average grades.

**Further Explorations:**

Now that you understand how to create and use Scalar Functions, you can further explore the possibilities:

```
-- Example: Use the Scalar Function in a WHERE clause
SELECT Name, Subject
FROM Teachers
WHERE dbo.GetAverageGrade(Subject) > 80;
```

**Explanation:**

- In this example, we use the Scalar Function dbo.GetAverageGrade(Subject) in a WHERE clause to filter the teachers based on their subject's average grade.
- The result set will contain the names and subjects of the teachers whose subject's average grade is greater than 80.

**Conclusion:**

Scalar Functions in T-SQL are a powerful tool for encapsulating reusable logic and returning a single value. In this lesson, we learned how to create and use Scalar Functions based on the previously provided table scripts. By incorporating Scalar Functions into your T-SQL code, you can modularize and reuse custom logic, making your queries more efficient and maintainable.

**Next Steps:**

Now that you have a good understanding of Scalar Functions, continue practicing by creating your own Scalar Functions. Explore different scenarios, such as performing calculations, applying custom business rules, or manipulating data. Keep experimenting and expanding your knowledge of T-SQL!

**Quiz**

What is the purpose of a Scalar Function in T-SQL?

- To encapsulate reusable logic and return a single value.
- To perform calculations on multiple values.
- To manipulate data in tables.
- To apply custom business rules.

What is the syntax for creating a Scalar Function?

- CREATE FUNCTION functionName(parameters) RETURNS returnType AS BEGIN...END;
- CREATE FUNCTION functionName(parameters) AS BEGIN...END;
- CREATE FUNCTION functionName(parameters) RETURNS dataType AS BEGIN...END;
- CREATE FUNCTION functionName(parameters) RETURN returnType AS BEGIN...END;

How can Scalar Functions be used in T-SQL queries?

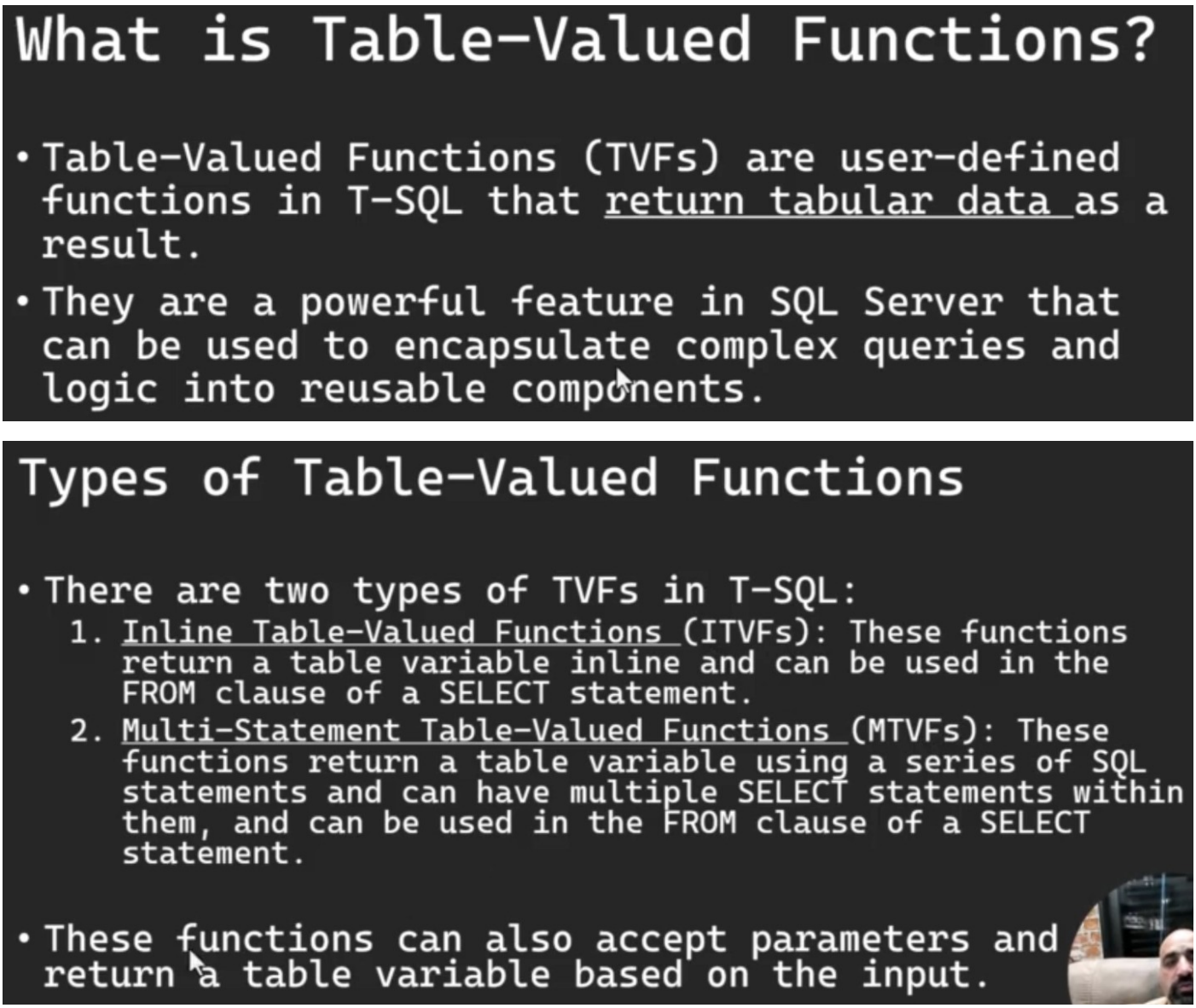
- In the SELECT statement only.
- In the WHERE clause only.
- In the FROM clause only.
- In any part of the query where an expression is allowed.

## Introduction

غير ال scalar function فيه عندك table valued functions ودي function بترجعلك جدول زي ال view كده وفيه منها نوعين

1- inline table valued function :- ودي بترجعلك جدول وتقدر تستعملها مع ال select statement عادي كأنها جدول مستقل بس بتستخدم select statement واحده جواها

2- multi statement table valued function :- ودي بترجعلك جدول برضه بس بتقدر تستخدم جواها اكتر من select statement




**Introduction:**

Table-Valued Functions (TVFs) are user-defined functions in T-SQL that return tabular data as a result. They are a powerful feature in SQL Server that can be used to encapsulate complex queries and logic into reusable components. In this lesson, we will explore the concept of TVFs, their types, and how to create and use them in T-SQL.

**Types of Table-Valued Functions:**

**There are two types of TVFs in T-SQL:**

1. **Inline Table-Valued Functions (ITVFs):** These functions return a table variable inline and can be used in the FROM clause of a SELECT statement.
2. **Multi-Statement Table-Valued Functions (MTVFs):** These functions return a table variable using a series of SQL statements and can have multiple SELECT statements within them.

These functions can also accept parameters and return a table variable based on the input.

## Quiz 1

What are Table-Valued Functions (TVFs) in T-SQL?

- Built-in functions in T-SQL that manipulate table data
- User-defined functions in T-SQL that return tabular data
- Predefined functions in T-SQL that return scalar values
- System functions in T-SQL that return metadata about tables

How many types of Table-Valued Functions (TVFs) are there in T-SQL?

- One type
- Two types
- Three types
- Four types

Where can Inline Table-Valued Functions (ITVFs) be used in T-SQL?

- In the SELECT clause
- In the WHERE clause
- In the FROM clause
- In the GROUP BY clause

Can Multi-Statement Table-Valued Functions (MTVFs) accept parameters?

- Yes
- No

Which type of Table-Valued Function can be used in the FROM clause of a SELECT statement?

- Inline Table-Valued Functions (ITVFs)
- Multi-Statement Table-Valued Functions (MTVFs)
- All of the above

Which type of Table-Valued Function can have multiple SELECT statements within them?

- Inline Table-Valued Functions (ITVFs)
- Multi-Statement Table-Valued Functions (MTVFs)
- Table-Valued Functions with Parameters
- None of the above

Which type of Table-Valued Function accepts parameters?

- Inline Table-Valued Functions (ITVFs)
- Multi-Statement Table-Valued Functions (MTVFs)
- All of the above

Table-Valued Function can be used with joins.

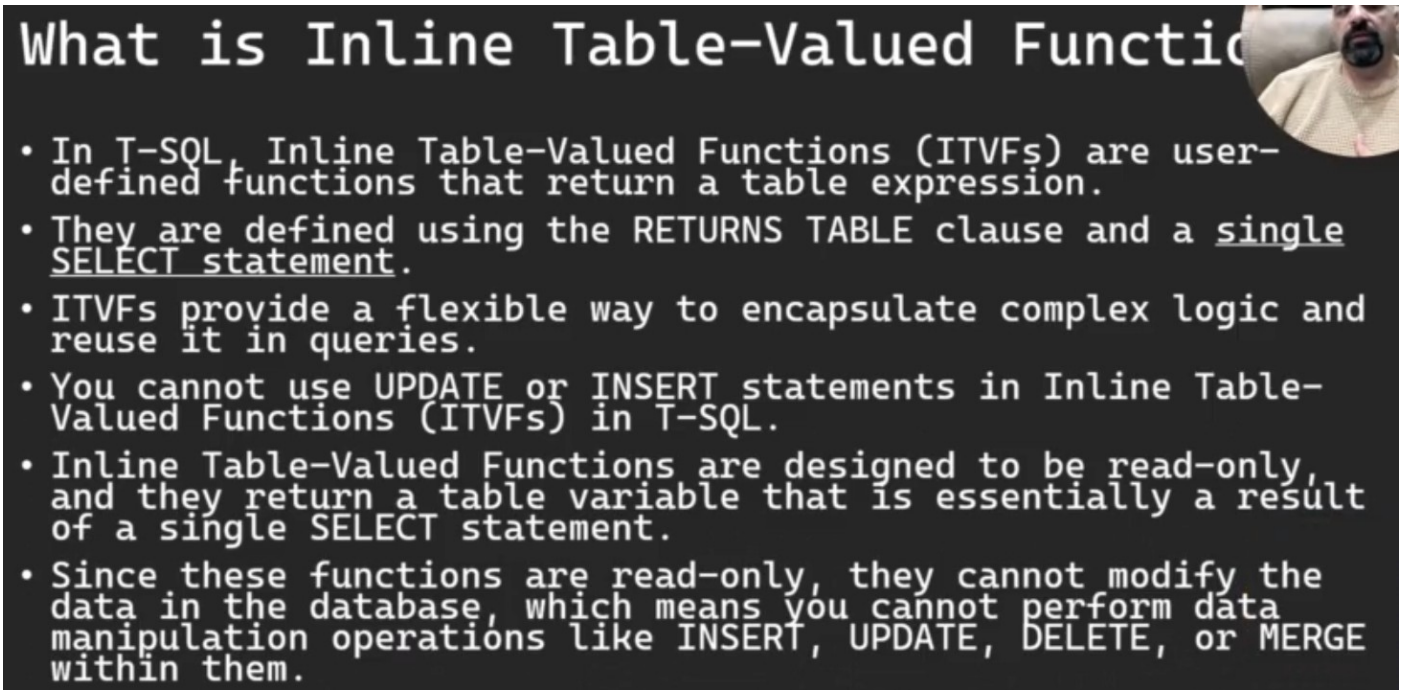
True

False

## Inline Table-Valued Functions (ITVFs) in T-SQL

زي ماقولنا قبل كده انه ال inline table valued function بترجعلك جدول وبتستخدم فيه single select statement

وخلي بالك انه كل ال functions بتكون read only




عشان تعرفها بتعرفها كانك بتعمل function عاديه بس ال return type بيكون table

كمان امر ال select اللي هترجع منه الداتا بتكتب return وتفتح قوسين تكتبه بينهم

تعالي اعملي function تاخد اسم الماده وترجع كل الطلاب اللي بتدرس الماده دي

```
create function GetStudentsBySubject(@Subject nvarchar(50))
returns table

as

return(
select * from Students
where Students.Subject=@Subject
)
```

وهنا بتستخدمها كانها جدول

```
select * from GetStudentsBySubject('Math`')
```

**Inline Table-Valued Functions (ITVFs) in T-SQL**

Understand the concept of Inline Table-Valued Functions (ITVFs) in T-SQL and learn how to create and use them effectively.

**Introduction:**
In T-SQL, Inline Table-Valued Functions (ITVFs) are user-defined functions that return a table expression. They are defined using the RETURNS TABLE clause and a single SELECT statement. ITVFs provide a flexible way to encapsulate complex logic and reuse it in queries.

You cannot use UPDATE or INSERT statements in Inline Table-Valued Functions (ITVFs) in T-SQL. Inline Table-Valued Functions are designed to be read-only, and they return a table variable that is essentially a result of a single SELECT statement. Since these functions are read-only, they cannot modify the data in the database, which means you cannot perform data manipulation operations like INSERT, UPDATE, DELETE, or MERGE within them.

The primary purpose of an ITVF is to encapsulate a `SELECT` query. This limitation ensures that the function remains deterministic and does not change the state of the database, which is a critical aspect for functions in SQL Server. If you need to perform insert or update operations, you would typically use Stored Procedures or other types of T-SQL routines that are designed for such purposes.

In this lesson, we will explore the concept of Inline Table-Valued Functions and demonstrate their usage based on the table script you provided.

**Understanding the Scenario:**
Before we dive into creating an Inline Table-Valued Function, let's understand the scenario based on the table script you provided.

**Table Script:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(50),
```

```
    Subject NVARCHAR(50),
    Grade INT
);
```

**Explanation:**

- The table script creates a table called Students with columns StudentID, Name, Subject, and Grade.
- The StudentID column is the primary key of the table.

**Creating an Inline Table-Valued Function:**

Now, let's create an Inline Table-Valued Function based on the scenario.

```
CREATE FUNCTION dbo.GetStudentsBySubject
(
    @Subject NVARCHAR(50)
)
RETURNS TABLE
AS
RETURN
(
    SELECT *
    FROM Students
    WHERE Subject = @Subject
)
```

**Explanation:**

- We create an Inline Table-Valued Function named "GetStudentsBySubject" that accepts a parameter called @Subject of type NVARCHAR(50).
- The RETURNS TABLE clause specifies that the function will return a table.
- Inside the function body, we use a SELECT statement to retrieve all the rows from the Students table where the Subject column matches the @Subject parameter value.

- The function ends with the RETURN statement, which returns the result set as the output of the function.

**Using the Inline Table-Valued Function:**

Now that we have created the Inline Table-Valued Function, let's see how we can use it in queries.

```
-- Example 1: Select all students studying Math
SELECT *
FROM dbo.GetStudentsBySubject('Math')



-- Example 2: Get the average grade of students studying Science
SELECT AVG(Grade)
FROM dbo.GetStudentsBySubject('Science')
```

**Explanation:**

- In Example 1, we use the SELECT statement to retrieve all the rows from the table-valued function dbo.GetStudentsBySubject('Math'). This will give us all the students studying Math.
- In Example 2, we calculate the average grade of students studying Science by using the AVG function and passing the table-valued function dbo.GetStudentsBySubject('Science') as the input.

**Conclusion:**

Inline Table-Valued Functions (ITVFs) in T-SQL provide a flexible way to encapsulate complex logic and reuse it in queries. They allow us to treat the function's output as a regular table, enabling powerful querying capabilities. In this lesson, we learned how to create an Inline Table-Valued Function based on the table script you provided and use it in queries. This knowledge will help you design and implement efficient database solutions in the future.

Next Steps:

Now that you have a good understanding of Inline Table-Valued Functions, try experimenting with different scenarios and explore more advanced topics, such as joining Inline Table-Valued Functions with other tables, applying filtering and aggregation functions on the results, and optimizing their performance. Keep practicing and exploring the world of T-SQL!

# Using Inline Table-Valued Functions with JOIN in T-SQL

هنا بيعرضلك مثال علي استخدام ال جدول الناتج عن ال function في ال join

فلو قولتلك هاتلي الطلاب اللي بيدرسوا مادة ال math وزود عمود حط فيه اسم المدرس بتاع الماده دي

```
use C21_DB1

select S.StudentID,S.Name,S.Grade,
Teachers.Name

from Teachers inner join dbo.GetStudentsBySubject('Math') S
on Teachers.Subject=S.Subject
```

لاحظ اننا ادينا اللي خارج من ال functionاسم عشان نعرف نستخدمه

**Using Inline Table-Valued Functions with JOIN in T-SQL**

Understand how to use Inline Table-Valued Functions with JOIN operations in T-SQL to combine data from multiple tables.

**Introduction:**

In T-SQL, Inline Table-Valued Functions (ITVFs) offer a powerful way to encapsulate complex logic and reuse it in queries. One of the key advantages of ITVFs is their ability to be used in JOIN operations, allowing us to combine data from multiple tables efficiently.

In this lesson, we will explore how to use Inline Table-Valued Functions with JOIN operations based on the previously created functions and tables.

**Recap of Previously Created Objects:**

Before we proceed, let's recap the previously created objects:

**Table Scripts:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT
);
```

```
CREATE TABLE Teachers (
    TeacherID INT PRIMARY KEY,
    Name NVARCHAR(50),
    Subject NVARCHAR(50)
);


Inline Table-Valued Functions:
CREATE FUNCTION dbo.GetStudentsBySubject
(
    @Subject NVARCHAR(50)
)
RETURNS TABLE
AS
RETURN
(
    SELECT *
    FROM Students
    WHERE Subject = @Subject
);
```

**Explanation:**

- We have two tables: Students and Teachers, which store information about students and teachers, respectively.
- We also have an Inline Table-Valued Function named "GetStudentsBySubject" that retrieves students based on their subjects.

**Using Inline Table-Valued Functions with JOIN:**

Now, let's see how we can use Inline Table-Valued Functions in JOIN operations to combine data from multiple tables.

```
-- Example 1: Join Students and Teachers based on shared subject
SELECT s.StudentID, s.Name AS StudentName, t.Name AS TeacherName, s.Grade
```

```
FROM dbo.GetStudentsBySubject('Math') s
JOIN Teachers t ON s.Subject = t.Subject
```

**Explanation:**

- In Example 1, we use the SELECT statement to retrieve data from the Students table-valued function dbo.GetStudentsBySubject('Math') and join it with the Teachers table based on the shared subject.
- We specify aliases for the Students and Teachers tables as "s" and "t", respectively, to make the query more readable.
- The JOIN keyword is used to combine the rows from both tables based on the specified condition, which is matching subjects in this case.
- The result set includes columns from both tables: StudentID, StudentName (alias for Name column in Students table), TeacherName (alias for Name column in Teachers table), and Grade.

**Conclusion:**

Using Inline Table-Valued Functions with JOIN operations allows us to combine data from multiple tables efficiently. In this lesson, we learned how to use Inline Table-Valued Functions in JOIN operations and explored different scenarios involving the Students and Teachers tables. By incorporating Inline Table-Valued Functions into your queries, you can enhance the flexibility and reusability of your T-SQL code.

**Next Steps:**

Now that you have a good understanding of using Inline Table-Valued Functions with JOIN operations, continue experimenting with different scenarios. Explore more advanced topics, such as combining multiple Inline Table-Valued Functions with JOINs, using different types of JOINs (e.g., LEFT JOIN, INNER JOIN), and incorporating additional filtering and aggregation functions. Keep practicing and expanding your knowledge of T-SQL!

**Quiz 2**

## What are Inline Table-Valued Functions (ITVFs) in T-SQL?

- They are user-defined functions that return a table expression.
- They are built-in functions in T-SQL.
- They are used to create temporary tables in T-SQL.
- They are used to update existing tables in T-SQL.

## How are Inline Table-Valued Functions defined?

- Using the RETURNS TABLE clause and a single SELECT statement.
- Using the CREATE TABLE statement.
- Using the CREATE FUNCTION statement.
- Using the SELECT INTO statement.

## What does the RETURNS TABLE clause specify?

- The name of the table to return.
- The columns to include in the output.
- The data type of the input parameter.
- The function will return a table.

## How do you use an Inline Table-Valued Function in a query?

- By calling the function and passing the appropriate parameters.
- By using the SELECT statement to retrieve data from the function.
- By using the INSERT statement to insert data into the function.
- By using the UPDATE statement to modify data in the function.

## What can you do with the output of an Inline Table-Valued Function?

- Treat it as a regular table and perform powerful querying operations.
- Use it to create a new table in the database.
- Use it to delete data from an existing table.
- Use it to update data in an existing table.

## What is an Inline Table-Valued Function (ITVF) in T-SQL?

- A function that returns multiple tables
- A function that returns a table variable and cannot have multiple statements in its body
- A function that returns a single scalar value
- A function that performs data manipulation operations

## Which of the following is a characteristic of an Inline Table-Valued Function in T-SQL?

- It can include multiple SQL statements
- It is composed of a single SELECT statement

How do you invoke an Inline Table-Valued Function in T-SQL?

- EXECUTE dbo.MyFunction()
- CALL dbo.MyFunction()
- SELECT * FROM dbo.MyFunction()
- dbo.MyFunction()

In an Inline Table-Valued Function in T-SQL, what must the RETURN statement contain?

- A variable
- A single SELECT statement
- A CREATE TABLE statement
- Multiple INSERT statements

Can i use update and insert statement in inline functions?

- No, you cannot use UPDATE or INSERT statements in Inline Table-Valued Functions (ITVFs) in T-SQL. Inline Table-Valued Functions are designed to be read-only, and they return a table variable that is essentially a result of a single SELECT statement. Since these functions are read-only, they cannot modify the data in the database, which means you cannot perform data manipulation operations like INSERT, UPDATE, DELETE, or MERGE within them.
- Yes

## Multi-Statement Table-Valued Functions (MTVFs) in T-SQL

ال ITVF يعتبر هوا هوا ال MTVF الفرق بينهم انه ال ITVF هوا بيستخدم SELECT STATEMENT واحده وال MTVF بتقدر تستخدم فيه اكتر من select statement

كمان فيه فرق انك في ال ITVF بترجع الجدول as generic table

انما في ال mtvf انت بترجع الجدول as variable table

يعني ايه؟

لما كنا بنعرف ال inline table valued function كنا بنكتب return table as وبعدها نحط ال select statement بتاعتنا فبيطلعلك أي جدول والسلام

انما هنا لا هنا انت لازم تعرف متغير من النوع table وتعرف مكوناته وبعد كده تتحكم فيه في انك تحذف او تضيف داتا وترجعه

بص الكود ده

ده ال itvf

```
CREATE FUNCTION dbo.GetStudentsBySubject
(
   @Subject NVARCHAR(50)
)
RETURNS TABLE
AS
RETURN
(
   SELECT *
   FROM Students
   WHERE Subject = @Subject
)
```

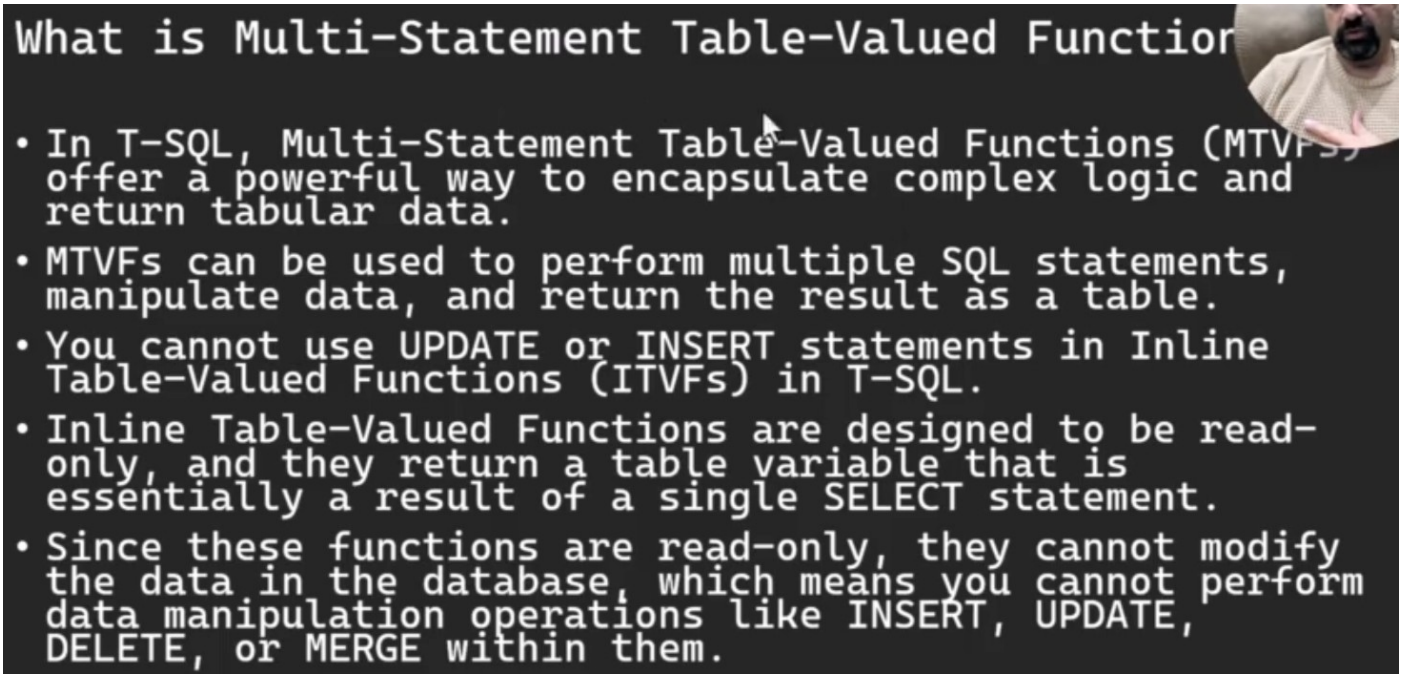
وده ال mtvf هتلاقيه جنب ال return بدل مايتكتب table وخلاص لا ده عرف متغير من النوع table

```
CREATE FUNCTION dbo.GetTopPerformingStudents()
RETURNS @Result TABLE (
   StudentID INT,
   Name NVARCHAR(50),
   Subject NVARCHAR(50),
   Grade INT
)
AS
BEGIN
   INSERT INTO @Result (StudentID, Name, Subject, Grade)
   SELECT TOP 3 StudentID, Name, Subject, Grade
   FROM Students
   ORDER BY Grade DESC;

   RETURN;
```

```
END;
```

وهنا بنستدعي ال function عادي

```
select * from dbo.GetTopPerformingStudents()
```




**Multi-Statement Table-Valued Functions (MTVFs) in T-SQL**

Understand how to create and use Multi-Statement Table-Valued Functions (MTVFs) in T-SQL to encapsulate complex logic and return tabular data.

**Introduction:**

In T-SQL, Multi-Statement Table-Valued Functions (MTVFs) offer a powerful way to encapsulate complex logic and return tabular data. MTVFs can be used to perform multiple SQL statements, manipulate data, and return the result as a table. This lesson will guide you through the process of creating and using MTVFs based on the previously provided table scripts.

**Recap of Previously Created Objects:**

Before we proceed, let's recap the previously created objects:

**Table Scripts:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
```

```
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT
);


CREATE TABLE Teachers (
    TeacherID INT PRIMARY KEY,
    Name NVARCHAR(50),
    Subject NVARCHAR(50)
);
```

**Explanation:**

- We have two tables: Students and Teachers, which store information about students and teachers, respectively.

**Creating a Multi-Statement Table-Valued Function:**

Now, let's create a Multi-Statement Table-Valued Function named "GetTopPerformingStudents".

```
CREATE FUNCTION dbo.GetTopPerformingStudents()
RETURNS @Result TABLE (
    StudentID INT,
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT
)
AS
BEGIN
    INSERT INTO @Result (StudentID, Name, Subject, Grade)
    SELECT TOP 3 StudentID, Name, Subject, Grade
    FROM Students
    ORDER BY Grade DESC;

    RETURN;
END;
```

Explanation:

- In the above script, we create a Multi-Statement Table-Valued Function named "GetTopPerformingStudents".
- The function returns a table with the columns StudentID, Name, Subject, and Grade.
- Inside the function, we declare a table variable called @Result to hold the result set.
- We use the INSERT INTO statement to populate the @Result table variable with the top 5 performing students from the Students table, ordered by Grade in descending order.
- Finally, we return the @Result table variable.

**Using the Multi-Statement Table-Valued Function:**

Now that we have created the Multi-Statement Table-Valued Function, let's use it in a query.

```
SELECT *
FROM dbo.GetTopPerformingStudents();
```

**Explanation:**

- We use the SELECT statement to retrieve data from the Multi-Statement Table-Valued Function dbo.GetTopPerformingStudents().
- The function is invoked like a table, so we can select all columns (*) from it.
- The result set will contain the top 3 performing students based on their grades.

**Further Explorations:**

Now that you understand how to create and use Multi-Statement Table-Valued Functions, you can further explore the possibilities:

```
-- Example: Join Teachers with the result of the Multi-Statement Table-Valued Function
SELECT t.Name AS TeacherName, s.Name AS StudentName, s.Grade
FROM Teachers t
JOIN dbo.GetTopPerformingStudents() s ON t.Subject = s.Subject
```

**Explanation:**

- In this example, we join the result of the Multi-Statement Table-Valued Function dbo.GetTopPerformingStudents() with the Teachers table.
- We join the tables based on the shared Subject column.
- The result set will contain the names of the teachers, the names of the top performing students, and their grades.

**Conclusion:**

Multi-Statement Table-Valued Functions (MTVFs) are a powerful tool for encapsulating complex logic and returning tabular data. In this lesson, we learned how to create and use MTVFs based on the previously provided table scripts. By incorporating MTVFs into your T-SQL code, you can modularize and reuse complex logic, making your queries more efficient and maintainable.

**Next Steps:**

Now that you have a good understanding of Multi-Statement Table-Valued Functions, continue practicing by creating your own MTVFs. Explore different scenarios, such as filtering data, performing calculations, or combining multiple tables. Keep experimenting and expanding your knowledge of T-SQL!

## Quiz 3

What is the purpose of Multi-Statement Table-Valued Functions (MTVFs) in T-SQL?

- To encapsulate complex logic and return tabular data.
- To perform single SQL statements and retrieve scalar values.
- To manipulate data in tables and modify table structures.
- To create temporary tables for storing intermediate results.

How is a Multi-Statement Table-Valued Function created in T-SQL?

- Using the CREATE TABLE statement
- Using the DECLARE statement
- Using the CREATE FUNCTION statement
- Using the INSERT INTO statement

Which of the following statements is true about MTVFs in T-SQL?

- They can modify the database state
- They are used for scalar computations only
- They must return a result set that is a table

How can you invoke a Multi-Table Valued Function in T-SQL?

- SELECT * FROM dbo.MyFunction()
- EXECUTE dbo.MyFunction()
- CALL dbo.MyFunction()
- dbo.MyFunction()

What must be included in the definition of a MTVF in T-SQL?

- An INSERT statement
- A RETURN statement with a table value
- A DELETE statement
- A transaction block

# Dynamic SQL

ال dynamic sql معناها انك بتبني ال query بتاعتك في شكل string اثناء ما البرنامج شغال وبعدين بتروح تنفذ ال query دي

تقدر تستخدم ال dynamic sql في ال stored procedures

بتنفذ ال dynamic sql ياما عن طريق الامر exec او sp_executesql

وبيقولك انه الحوار ده بيديك مرونه اكتر في التعامل بس لازم تكون مصحصح وانت بتعمله والأستاذ مش بيفضل استخدامه لانه ال debugging بيكون اصعب  وال performance بيكون ابطأ وكمان ال security بتكون اقل

دول مثالين عليه

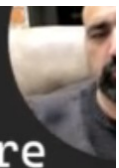
```
CREATE PROCEDURE GenerateDynamicSQL1
   @TableName NVARCHAR(128)
AS
BEGIN
   DECLARE @SQL NVARCHAR(MAX);
   SET @SQL = 'SELECT * FROM ' + @TableName;
   EXECUTE(@SQL);
END

CREATE PROCEDURE GenerateDynamicSQL2
   @TableName NVARCHAR(128)
AS
BEGIN
   DECLARE @SQL NVARCHAR(MAX);
   SET @SQL = N'SELECT * FROM ' + QUOTENAME(@TableName);
   EXEC sp_executesql @SQL;
END
```

وده عن طريقة استخدامهم

```
USE [C21_DB1]
GO

DECLARE @RC int
DECLARE @TableName nvarchar(128)

-- TODO: Set parameter values here.
```

```
set @TableName='Students';

EXECUTE @RC = [dbo].[GenerateDynamicSQL1]
  @TableName


EXECUTE @RC = [dbo].[GenerateDynamicSQL2]
  @TableName

GO
```

## What is Dynamic SQL



- Dynamic SQL refers to SQL statements that are constructed and executed at runtime as strings.
- Stored procedures in SQL Server can be used to generate and execute dynamic SQL.
- Dynamic SQL Execution: Using EXEC or sp_executesql to execute dynamically built SQL strings.
- This approach allows for a high degree of flexibility, but it also requires careful handling to avoid issues such as SQL injection.

## Considerations for Dynamic SQL in Store Procedures

- Debugging and Maintenance: Dynamic SQL can be hard to debug and maintain compared to static SQL, as the actual SQL executed is constructed at runtime.
- Performance: While dynamic SQL offers flexibility, it may not always be as performant as static SQL, especially if it leads to frequent recompilations of the SQL statements.
- Security: Ensure that the stored procedure and its dynamic SQL do not inadvertently elevate the privileges of the user executing it.
- SQL Injection: Dynamic SQL can be vulnerable to SQL injection attacks, especially if user input is concatenated directly into SQL strings. Always validate and sanitize user inputs.

Stored procedures in SQL Server can be used to generate and execute dynamic SQL. Dynamic SQL refers to SQL statements that are constructed and executed at runtime as strings. This approach allows for a high degree of flexibility, but it also requires careful handling to avoid issues such as SQL injection.

## Generating Dynamic SQL in Stored Procedures

To generate and execute dynamic SQL within a stored procedure, you can use the `EXECUTE` statement or the `sp_executesql` system stored procedure. Here's how each method works:

- Using `EXECUTE` (or `EXEC`):

```
CREATE PROCEDURE GenerateDynamicSQL1
    @TableName NVARCHAR(128)
AS
BEGIN
    DECLARE @SQL NVARCHAR(MAX);
    SET @SQL = 'SELECT * FROM ' + @TableName;
    EXECUTE(@SQL);
END
```

    - 
    - This method directly executes a dynamically built SQL string.
    - Be cautious of SQL injection risks when concatenating strings to build the SQL statement.

- Using `sp_executesql`:

```
CREATE PROCEDURE GenerateDynamicSQL2
    @TableName NVARCHAR(128)
AS
BEGIN
    DECLARE @SQL NVARCHAR(MAX);
    SET @SQL = N'SELECT * FROM ' + QUOTENAME(@TableName);
    EXEC sp_executesql @SQL;
END
```

- 
    - `sp_executesql` supports parameterized queries, which can help mitigate the risk of SQL injection.
    - The `QUOTENAME` function is used to safely escape the table name, providing additional protection against injection.

## Considerations for Dynamic SQL in Stored Procedures

- SQL Injection: Dynamic SQL can be vulnerable to SQL injection attacks, especially if user input is concatenated directly into SQL strings. Always validate and sanitize user inputs.
- Debugging and Maintenance: Dynamic SQL can be harder to debug and maintain compared to static SQL, as the actual SQL executed is constructed at runtime.
- Performance: While dynamic SQL offers flexibility, it may not always be as performant as static SQL, especially if it leads to frequent recompilations of the SQL statements.
- Security: Ensure that the stored procedure and its dynamic SQL do not inadvertently elevate the privileges of the user executing it.

In summary, while stored procedures can be used to generate and execute dynamic SQL in SQL Server, it's important to use this feature judiciously and securely, considering the implications for security, performance, and maintainability.

## SQL injection attack

ال sql injection دي بتبقي مشكله في ال security عشان بتستخدم dynamic sql

الفكره هنا انك لما بتكتب ال dynamic sql ممكن حد يضيف شرط صغير يرجع الداتا كلها اللي هوا ينفع يشوفها واللي ماينفعش يشوفها

والمصيبه الأكبر انه يعملهالك في update statement او delete هتلاقي نفس بتقول شقي عمري راح

والأفضل انك تكتب ال query بتاعتك عادي وتمرر بس القيم اللي عايز تمررها عن طريق ال parameters

بص كده علي دي

هنا الأوامر دي بترجعلك الطالب اللي رقمه 1

```
DECLARE @input NVARCHAR(50) = '1';


DECLARE @SQL NVARCHAR(MAX);


SET @SQL =  ' SELECT * FROM Students WHERE studentID = ' +
@input;
EXECUTE(@SQL);
```

تعالي ضيف شرط صغير وهتلاقي نفسك شايف البحر من هنا

```
DECLARE @input NVARCHAR(50) = '1 or 1=1';


DECLARE @SQL NVARCHAR(MAX);


SET @SQL =  ' SELECT * FROM Students WHERE studentID = ' +
@input;
EXECUTE(@SQL);
```

وده الأفضل انك تخلي ال query زي ماهي وال parameters بس هيا اللي تتغير

```
DECLARE @input NVARCHAR(50) = '1 or 1=1';

SELECT *
FROM Students
WHERE studentID = @input;
```

**SQL Injection in T-SQL**

Understand the concept of SQL Injection, its potential risks, and learn preventive measures to mitigate this security vulnerability in T-SQL using the provided script.

**Introduction:**

SQL Injection is a security vulnerability that occurs when an attacker manipulates user input to execute unintended SQL statements. This can lead to unauthorized access, data breaches, or even

complete compromise of the database. In this lesson, we will explore SQL Injection risks and learn preventive measures using the provided script as an example.

**Understanding SQL Injection:**
SQL Injection occurs when an attacker injects malicious SQL code into an application's input fields, bypassing input validation and directly manipulating SQL statements. Let's consider the provided script as an example:

```
DECLARE @input NVARCHAR(50) = '1 or 1=1';


DECLARE @SQL NVARCHAR(MAX);


SET @SQL =  ' SELECT * FROM Students WHERE studentID = ' +  @input;
EXECUTE(@SQL);
```

**Explanation:**

- In this example, the @input variable is set to `'1 or 1=1'`, which is a typical SQL Injection payload.
- The script dynamically creates a SQL statement in the @SQL variable by concatenating the @input value into the query.
- If the @input value is not properly validated or sanitized, an attacker can manipulate it to alter the original query's logic and retrieve unintended data.

**SQL Injection Risks:**
**SQL Injection poses several risks, including:**

- Unauthorized data access: Attackers can retrieve sensitive information by crafting malicious input.
- Data modification: Attackers can modify or delete data in the database.
- Database compromise: Attackers can execute arbitrary SQL statements, potentially gaining full control of the database.

**Preventive Measures:**

**To prevent SQL Injection, follow these best practices:**

**Parameterized Queries:**

- 
  - Use parameterized queries or prepared statements instead of concatenating user input directly into SQL statements.
  - Parameterized queries separate the SQL code from the user input, preventing malicious input from altering the query's structure.

Example using parameterized queries:

```
DECLARE @input NVARCHAR(50) = '1 or 1=1';



SELECT *
FROM Students
WHERE studentID = @input;
```

Explanation:

- In this example, the @input value is passed as a parameter to the query, separating it from the SQL code.
- Even if the @input value is `'1 or 1=1'`, it will be treated as a literal value and not alter the query's structure.

**Input Validation and Sanitization:**

- Validate and sanitize user input to ensure it meets expected criteria.
- Use input validation techniques such as white-listing, regular expressions, or data type checks to filter out potentially malicious input.

**Example using input validation:**

```
DECLARE @input NVARCHAR(50) = '1 or 1=1';
```

```
IF ISNUMERIC(@input) = 1
BEGIN
    SELECT *
    FROM Students
    WHERE studentID = @input;
END
ELSE
BEGIN
    -- Handle invalid input
    SELECT 'Invalid input';
END
```

**Explanation:**

- In this example, the ISNUMERIC function is used to check if the @input value is numeric.
- If the @input value is numeric, the query is executed, ensuring that only valid studentIDs are used.
- Otherwise, an alternative action can be taken, such as displaying an error message.

**Conclusion:**

SQL Injection is a severe security vulnerability that can lead to unauthorized access, data breaches, or even complete compromise of the database. In this lesson, we explored the concept of SQL Injection, its potential risks, and learned preventive measures using the provided script as an example. By employing parameterized queries and implementing input validation and sanitization techniques, you can significantly reduce the risk of SQL Injection attacks.

**Quiz**

## What is dynamic SQL?

- SQL statements that are constructed and executed at runtime as strings
- SQL statements that are written in a stored procedure
- SQL statements that are executed using the EXECUTE statement
- SQL statements that are generated by the sp_executesql system stored procedure

## Which method can be used to generate and execute dynamic SQL using a stored procedure?

- EXECUTE statement
- sp_executesql system stored procedure
- CREATE PROCEDURE statement
- SELECT statement

## Which function is used to safely escape a table name in dynamic SQL?

- QUOTENAME
- CONCAT
- REPLACE
- CHARINDEX

## Why is dynamic SQL harder to debug and maintain compared to static SQL?

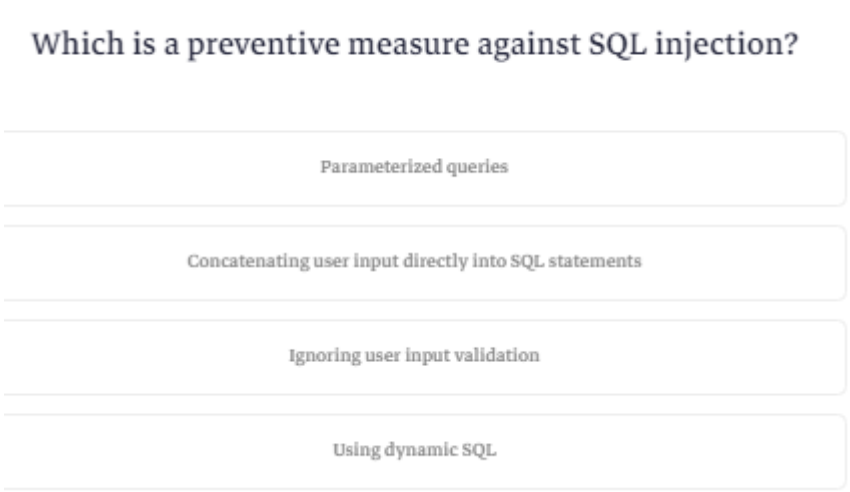
- The actual SQL executed is constructed at runtime
- Dynamic SQL requires the use of loops
- Dynamic SQL is not supported by SQL Server
- Static SQL is more prone to syntax errors

## Which of the following best describes SQL injection?

- A security vulnerability that occurs when an attacker manipulates user input to execute unintended SQL statements
- A feature in SQL Server that allows for the generation and execution of dynamic SQL
- A technique used to improve the performance of SQL queries
- A method of storing and retrieving data in a relational database

## What is the risk of SQL injection?

- Unauthorized data access
- Increased performance of SQL queries
- Improved database security
- Easier debugging of SQL statements




## Introduction to Triggers in T-SQL

ال triggers زيها زي ال events في ال #c كده لما تحصل حاجه معينه بتشغل function

وتقدر تنفذها علي table او علي view

تقدر تستخدمها مثلا لو حصل update او delete ل record معين تقوله يشغل ال procedure دي

وفيه منها أنواع زي ال after trigger وده بيشتغل بعد ماتنفذ عملية ال update او ال delete او ال insert

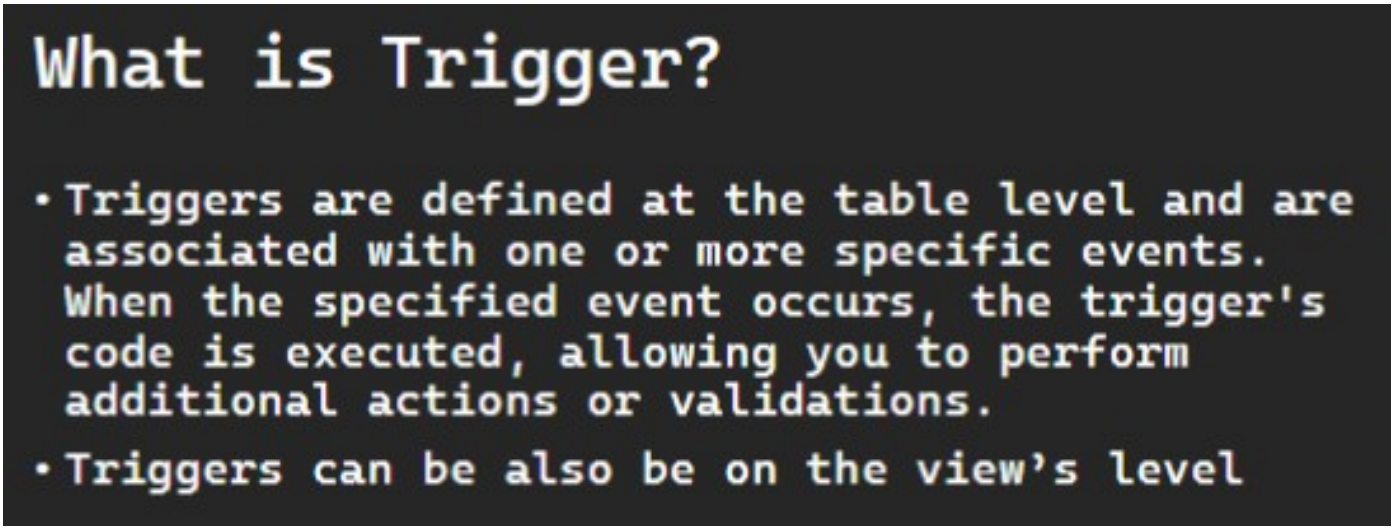
ونوع تاني اسمه instead of triggers وده بدل ماتخلي ال sql هوا اللي ينفذ العمليه لا انت بتكتب الكود اللي بينفذها بنفسك

ال triggers مكونه من 3 حاجات

ال trigger event عاوز تنفذ الكود بتاعك امتي

ال trigger condition لو حصلت حاجه اعملي حاجه

ال trigger action الكود اللي عايز تنفذه

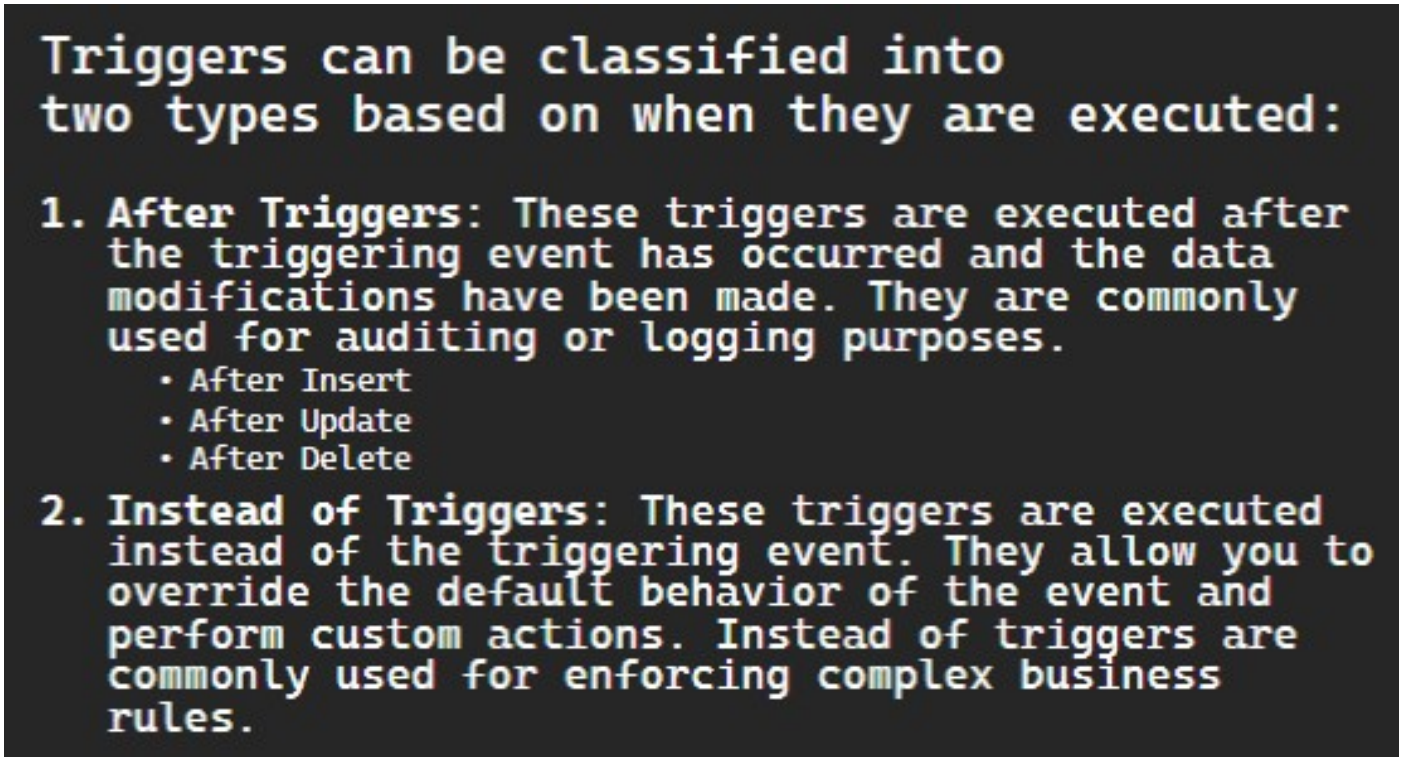


Triggers can be useful in various scenarios, such as:

- Enforcing business rules: Triggers can enforce complex business rules by validating data before allowing it to be inserted, updated, or deleted.
- Auditing and logging: Triggers can be used to track changes made to database tables, capturing information like who made the change, when it occurred, and what data was modified.
- Data synchronization: Triggers can be used to synchronize data between tables or databases in real-time, ensuring consistency across systems.
- Automatic updates: Triggers can automatically update related tables or fields when a specific event occurs, simplifying data maintenance.

## Triggers consist of three main components:

- Trigger Event: Specifies the event or events that will activate the trigger, such as INSERT, UPDATE, DELETE, or table-level events.
- Trigger Condition: Defines the condition that must be met for the trigger to execute. This condition can be based on specific column values or other criteria.
- Trigger Action: Contains the code or actions that will be executed when the trigger is activated. This can include SQL statements, stored procedure calls, or other operations.




**Introduction:**

Triggers in T-SQL are special types of stored procedures that are automatically executed in response to specific events occurring in the database. These events can include data modifications, such as INSERT, UPDATE, or DELETE operations, or database-level events like table creations or modifications.

Triggers are defined at the table level and are associated with one or more specific events. When the specified event occurs, the trigger's code is executed, allowing you to perform additional actions or validations.

**Triggers can be useful in various scenarios, such as:**

1. Enforcing business rules: Triggers can enforce complex business rules by validating data before allowing it to be inserted, updated, or deleted.
2. Auditing and logging: Triggers can be used to track changes made to database tables, capturing information like who made the change, when it occurred, and what data was modified.
3. Data synchronization: Triggers can be used to synchronize data between tables or databases in real-time, ensuring consistency across systems.
4. Automatic updates: Triggers can automatically update related tables or fields when a specific event occurs, simplifying data maintenance.

**Triggers in T-SQL are created using the CREATE TRIGGER statement and consist of three main components:**

1. Trigger Event: Specifies the event or events that will activate the trigger, such as INSERT, UPDATE, DELETE, or table-level events.
2. Trigger Condition: Defines the condition that must be met for the trigger to execute. This condition can be based on specific column values or other criteria.
3. Trigger Action: Contains the code or actions that will be executed when the trigger is activated. This can include SQL statements, stored procedure calls, or other operations.

**Triggers can be classified into two types based on when they are executed:**

1. **After Triggers:** These triggers are executed after the triggering event has occurred and the data modifications have been made. They are commonly used for auditing or logging purposes.
2. **Instead of Triggers:** These triggers are executed instead of the triggering event. They allow you to override the default behavior of the event and perform custom actions. Instead of triggers are commonly used for enforcing complex business rules.

It's important to use triggers judiciously as they can impact performance and introduce additional complexity to database operations. Proper testing and monitoring are essential when working with triggers to ensure they function as intended and do not cause any unintended side effects.

## Quiz

What is a trigger in T-SQL?

- A special type of stored procedure
- A type of database-level event
- A way to update related tables automatically
- A type of business rule enforcement

## In T-SQL, triggers are associated with:

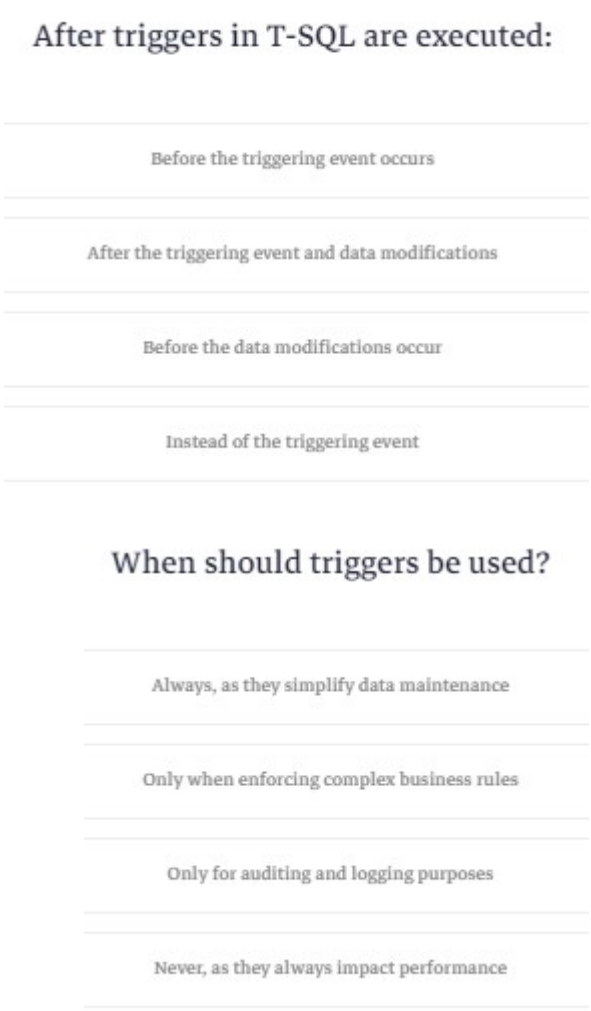
- Specific columns
- Specific events
- Specific databases
- Specific tables

## What are some scenarios where triggers can be useful?

- Enforcing business rules
- Auditing and logging
- Data synchronization
- All of the above

## How are triggers created in T-SQL?

- Using the CREATE TRIGGER statement
- Using the CREATE PROCEDURE statement
- Using the CREATE EVENT statement
- Using the CREATE TABLE statement




Instead of Triggers: These triggers are executed instead of the triggering event. They allow you to override the default behavior of the event and perform custom actions.

True

False

## After Insert Triggers in T-SQL

ال after insert trigger هوا event بيتم استدعاؤه لما بيحصل insert

ده جدول عملناه عشان نخزن فيه أي عملية insert جديدة تحصل في جدول ال students

```
use C21_DB1
create table StudentInsertLog(
LogID int identity Primary Key,
StudentID int,
Name nvarchar(50),
```

```
Subject nvarchar(50),
Grade int,
InsertedDateTime datetime default getdate()
)
```

طيب عايزين دلوقتي نعمل ال trigger اللي هيخزن الداتا في الجدول ده

هنعمله ازاي؟

بتكتب create trigger وبعديه اسمه وبعدين on وبعديه اسم الجدول او ال view اللي عايز تنفذ عليه الكود بتاعك وبعديه نوع ال trigger اللي عايز تعمله في حالتنا هنا هنكتب after insert وبعدين as وبعدين begin و end وفي وسطهم الكود اللي انت عايز تعمله

```
create trigger trg_AfterInsertStudent on Students
after insert
begin

end
```

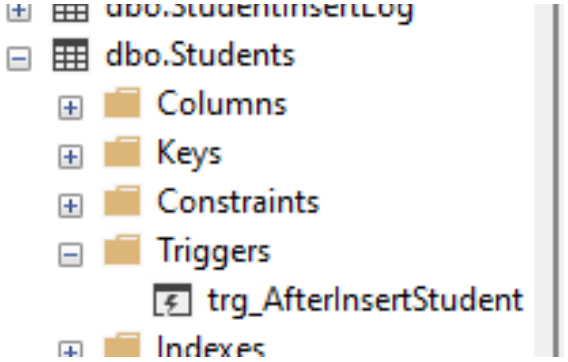
طيب الداتا بتاعت ال record اللي دخلناه هنجيبها منين؟

في ال triggers بيعملك جدولين رئيسين عشان تتابع منهم الداتا بتاعتك واحد اسمه inserted بيتحط فيه ال records اللي انضافت وبيتم استخدامه في ال update كمان

الجدول التاني اسمه deleted بيتحط فيه الداتا اللي اتحذفت او الداتا قبل التعديل

```
create trigger trg_AfterInsertStudent on Students
after insert
as
begin
insert into StudentInsertLog(StudentID ,Name ,Subject
,Grade )
select StudentID,Name,Subject,Grade from inserted
end
```

وده مكانه اهو




تعالي ندخل record جديد علي جدول ال students

```
insert into Students(StudentID,Name,Subject,Grade)
values(10,'Ali Doe','Math',85)
```

تعالي بقي نبص عالجدولين

```
select * from Students
select * from StudentInsertLog
```

## Understanding and Implementing After Insert Triggers in T-SQL

This lesson provides a detailed understanding of creating and testing an After Insert Trigger in T-SQL, using the 'Students' table as a practical example. We'll focus on how this trigger can be used to log new entries in a database.

**What is an After Insert Trigger?**

An After Insert Trigger in T-SQL is executed automatically after a new record is inserted into a table. It's typically used for tasks that need to occur immediately following the insertion, such as logging, auditing, or updating other related data.

**Environment Setup:**

Ensure the primary table ('Students') and a logging table ('StudentInsertLog') are in place.

```
-- Assuming the Students table is already created
-- Table for logging new student entries
CREATE TABLE StudentInsertLog (
    LogID INT IDENTITY PRIMARY KEY,
    StudentID INT,
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT,
```

```
    InsertedDateTime DATETIME DEFAULT GETDATE()
);
```

### Creating the After Insert Trigger:

The following script creates a trigger that activates after a new student record is inserted.

```
CREATE TRIGGER trg_AfterInsertStudent ON Students
AFTER INSERT
AS
BEGIN
    INSERT INTO StudentInsertLog(StudentID, Name, Subject, Grade)
    SELECT StudentID, Name, Subject, Grade FROM inserted;
END;
```

### Key Components Explained:

- `CREATE TRIGGER trg_AfterInsertStudent`: Declares a new trigger named `trg_AfterInsertStudent`.
- `ON Students`: Specifies that the trigger is associated with the 'Students' table.
- `AFTER INSERT`: Indicates that the trigger responds to insert operations.
- `INSERT INTO StudentInsertLog(...) SELECT ... FROM inserted`: The core action, which inserts a record into the 'StudentInsertLog' table. The `inserted` table is a special table used within the scope of insert and update triggers, holding the newly inserted rows.

### Testing the Trigger:

To validate our trigger, we insert a record into 'Students' and then query 'StudentInsertLog'.

```
-- Inserting a new student
INSERT INTO Students (StudentID, Name, Subject, Grade)
VALUES (1, 'John Doe', 'Mathematics', 85);



-- Checking the log table
```

```
SELECT * FROM StudentInsertLog;
```

**Expected Result:**

You should see a corresponding entry for 'John Doe' in the `StudentInsertLog` table.

**Conclusion:**

After Insert Triggers are powerful for maintaining data integrity, automating logging, and implementing complex business rules. They help ensure that actions are taken immediately after data is entered into the database, enhancing the database's functionality and reliability.

## Quiz 1

What is an After Insert Trigger in T-SQL?

- It is a trigger that is executed automatically after a record is deleted from a table.
- It is a trigger that is executed automatically after a record is updated in a table.
- It is a trigger that is executed automatically after a record is inserted into a table.

What are the benefits of After Insert Triggers?

- Maintaining data integrity and reliability in a database.
- Automating logging of new entries in a database.
- Implementing complex business rules in a database.
- All of the above.

## After Update Triggers in T-SQL

هناخد ال after update trigger

تعالي نعمل الجدول اللي هنسجل فيه الداتا قبل التعديل الأول

```sql
create table AfterUpdateLog(
LogID int identity primary key,
StudentID int,
OldGrade int,
NewGrade int,
UpdatedDateTime datetime default getdate()
)
```

تعالي نعمل الtrigger

الفكره هنا انه بيقوله لو حصل update علي عمود ال grade بيعمل insert وبيجيب الداتا عن طريق join مابين جدول ال inserted وجدول ال deleted

```sql
create trigger trg_AfterUpdateStudent on Students
after update
as
begin
if Update(Grade)
insert into AfterUpdateLog(StudentID,OldGrade,NewGrade)
select i.StudentID,d.Grade,i.Grade
from inserted i inner join deleted d
on i.StudentID=d.StudentID
end
```

```sql
update Students set Grade=100
where StudentID=10

select * from Students
select * from AfterUpdateLog
```

## Understanding After Update Triggers in T-SQL

This lesson aims to deepen your understanding of After Update Triggers in T-SQL, demonstrated with the 'Students' table. We will explore how these triggers can be used to monitor and log updates.

**What is an After Update Trigger?**

An After Update Trigger in T-SQL is a procedural code that is automatically executed in response to an UPDATE event on a table. It's mainly used for operations that need to occur right after a record is updated, like logging changes, enforcing constraints, or updating related information.

**Environment Setup:**

Ensure both the main table ('Students') and the update log table ('StudentUpdateLog') are prepared.

```
-- Assuming the Students table is already created



-- Table for logging grade updates
CREATE TABLE StudentUpdateLog (
    LogID INT IDENTITY PRIMARY KEY,
    StudentID INT,
    OldGrade INT,
    NewGrade INT,
    UpdatedDateTime DATETIME DEFAULT GETDATE()
);
```

**Creating the After Update Trigger:**

This script sets up a trigger to record grade changes in the 'Students' table.

```
CREATE TRIGGER trg_AfterUpdateStudent ON Students
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Grade)
    BEGIN
        INSERT INTO StudentUpdateLog(StudentID, OldGrade, NewGrade)
        SELECT i.StudentID, d.Grade AS OldGrade, i.Grade AS NewGrade
        FROM inserted i
        INNER JOIN deleted d ON i.StudentID = d.StudentID;
    END
END;
```

**Key Components Explained:**

- `CREATE TRIGGER trg_AfterUpdateStudent`: Creates a trigger named `trg_AfterUpdateStudent`.

- `ON Students`: Associates the trigger with the 'Students' table.
- `AFTER UPDATE`: The trigger is fired after an update operation.
- `IF UPDATE(Grade)`: Checks if the 'Grade' column was part of the update.
- The `inserted` and `deleted` tables: Special tables where `inserted` contains the new values and `deleted` contains the old values of the updated rows.

**Testing the Trigger:**

We'll update a student's grade and then inspect the update log.

```
-- Updating the grade of a student
UPDATE Students
SET Grade = 90
WHERE StudentID = 1;



-- Checking the log table
SELECT * FROM StudentUpdateLog;
```

**Expected Result:**

An entry in `StudentUpdateLog` showing the change from the old grade (85) to the new grade (90).

**Conclusion:**

After Update Triggers in T-SQL are essential for tracking modifications, enforcing data consistency, and automating responses to data changes. They play a critical role in maintaining the integrity and reliability of the data within a SQL Server database.

Through these lessons, you've gained valuable insights into implementing and utilizing After Insert and After Update Triggers, enhancing your T-SQL programming skills for effective database management.

**Quiz 2**

## What is an After Update Trigger in T-SQL?

- A procedural code that is automatically executed in response to an UPDATE event on a table.
- A command that updates all records in a table.
- A function that returns the maximum value in a column.
- A query that retrieves data from multiple tables.

## What are some common uses of After Update Triggers in T-SQL?

- Logging changes
- Updating related information
- Enforcing constraints
- All of the above

## Which tables are used in an After Update Trigger?

- Updated
- Deleted
- Inserted
- All of the above

When is an After Update Trigger fired?

Before an update operation

After an update operation

During an update operation

None of the above

## After Delete Triggers in T-SQL

هناخد ال after delete trigger

هنعمل جدول نحط فيه كل الstrudents اللي اتحذفوا

```
create table AfterDeleteLog(
LogID int identity primary key,
StudentID int,
Name nvarchar(50),
Subject nvarchar(50),
Grade int,
DeletedDateTime datetime  default getdate()
)
```

تعالي نعمل ال trigger

```
create trigger trg_FterDeleteStudent on Students
after delete

as
begin
insert into AfterDeleteLog(StudentID,Name,Subject,Grade)
select StudentID,Name,Subject,Grade from deleted
end
```

```
delete Students where StudentID=10

select * from Students
select * from AfterDeleteLog
```

# Understanding and Implementing After Delete Triggers in T-SQL

This lesson covers the creation and implementation of an After Delete Trigger in T-SQL, using the 'Students' table as a practical example. We aim to demonstrate how to use such triggers for logging and auditing purposes when records are deleted from a table.

### What is an After Delete Trigger?

An After Delete Trigger in T-SQL is a special kind of stored procedure that automatically executes after a DELETE operation is performed on a table. This trigger is commonly used for tasks like logging deletions, enforcing business rules, or maintaining referential integrity.

### Preparing the Logging Table

Before creating the trigger, we need a table to log the deletions. Here, we create a `StudentDeleteLog` table to store information about each deleted student record.

```
CREATE TABLE StudentDeleteLog (
    LogID INT IDENTITY PRIMARY KEY,
    StudentID INT,
    Name NVARCHAR(50),
    Subject NVARCHAR(50),
    Grade INT,
    DeletedDateTime DATETIME DEFAULT GETDATE()
);
```

This table includes columns to store the student's details and the time of deletion.

### Creating the After Delete Trigger

Now, we'll create the trigger that logs every deletion from the `Students` table.

```
CREATE TRIGGER trg_AfterDeleteStudent ON Students
AFTER DELETE
AS
BEGIN
    INSERT INTO StudentDeleteLog(StudentID, Name, Subject, Grade)
    SELECT StudentID, Name, Subject, Grade FROM deleted;
```

```
END;
```

**Key Components Explained:**

- `CREATE TRIGGER trg_AfterDeleteStudent ON Students`: This statement creates a new trigger named `trg_AfterDeleteStudent` on the `Students` table.
- `AFTER DELETE`: Specifies that the trigger responds to DELETE operations.
- The trigger's action: Inserts a record into the `StudentDeleteLog` table, selecting data from the `deleted` table, which is a special table containing the affected rows during the DELETE operation.

**Testing the Trigger**

To see our trigger in action, we'll delete a student record and then check the `StudentDeleteLog` table.

```
-- Assuming there is a student with StudentID = 1 in the Students table



-- Deleting a student
DELETE FROM Students WHERE StudentID = 1;



-- Checking the delete log table
SELECT * FROM StudentDeleteLog;
```

**Expected Result:**

The `StudentDeleteLog` table should have an entry corresponding to the deleted student, capturing their details at the time of deletion.

**Conclusion:**

After Delete Triggers are crucial in scenarios where tracking deletions is important, such as for auditing, maintaining historical data, or ensuring data integrity post-deletion. Through this lesson, you've learned how to effectively implement and test an After Delete Trigger in T-SQL, a skill that enhances data management and security in your SQL Server databases.

## Quiz 3

### What is an After Delete Trigger?

- A stored procedure that automatically executes after an UPDATE operation
- A special kind of stored procedure that automatically executes after a DELETE operation
- A trigger that only executes if there are errors during a DELETE operation
- A trigger that only executes if a rollback is performed

### What does the StudentDeleteLog table store?

- Information about each inserted student record
- Information about each updated student record
- Information about each deleted student record
- Information about each student's grades

### When does the trg_AfterDeleteStudent trigger execute?

- Before a DELETE operation
- During a DELETE operation
- After a DELETE operation
- Only if there are errors during a DELETE operation

# Inserted and Deleted Tables in T-SQL Triggers

هنا بيراجع معاك علي الجدولين بتوع ال INSERTED  وال DELETED اللى بيتعملوا اثناء ال TRIGGER

## Understanding the `inserted` and `deleted` Tables in T-SQL Triggers

In this lesson, we will delve into two special tables in T-SQL - `inserted` and `deleted` - that play a crucial role in the functionality of triggers. These tables are automatically created and managed by SQL Server during the execution of triggers associated with Data Manipulation Language (DML) operations like INSERT, UPDATE, and DELETE.

By the end of this lesson, you will understand how the `inserted` and `deleted` tables are used in T-SQL triggers to capture data changes, and how they can be leveraged in various scenarios, particularly using our example `Students` table.

**What are `inserted` and `deleted` Tables?**

- `inserted` Table: In the context of an INSERT or UPDATE operation, the `inserted` table contains the new version of each row that has been inserted or updated.
- `deleted` Table: During DELETE or UPDATE operations, the `deleted` table holds the original version of each row that has been deleted or updated.

These tables are fundamental in understanding and responding to data changes within triggers.

**Key Characteristics:**

- They are virtual tables, meaning they are not physically stored in the database but are created temporarily during trigger execution.
- Their structure mirrors the table on which the trigger is defined, containing all columns of the original table.
- They are essential for writing logic inside triggers that respond to data changes.

**Usage in Different DML Operations**:

- **INSERT**:

- `inserted` contains the new data being inserted.
- `deleted` is not used as no data is being removed.

- **DELETE:**
  - `deleted` contains the data being removed.
  - `inserted` is not used as no new data is being added.

- **UPDATE:**
  - `inserted` holds the updated (new) version of the data.
  - `deleted` holds the original data before the update.

### Example Scenario: Trigger on the `Students` Table

### Consider our `Students` table:

```
CREATE TABLE [dbo].[Students](
    [StudentID] [int] NOT NULL,
    [Name] [nvarchar](50) NULL,
    [Subject] [nvarchar](50) NULL,
    [Grade] [int] NULL,
    [IsActive] [bit] NULL,
    PRIMARY KEY CLUSTERED ([StudentID] ASC)
);
```

Scenario: We want to track changes in the `Grade` of students. For this, we create a trigger that logs any grade changes to an audit table.

### Step 1: Creating an Audit Table

```
CREATE TABLE StudentGradeAudit (
    AuditID INT IDENTITY PRIMARY KEY,
    StudentID INT,
    OldGrade INT,
    NewGrade INT,
    ChangeDateTime DATETIME DEFAULT GETDATE()
);
```

### Step 2: Creating the Trigger

```
CREATE TRIGGER trg_StudentGradeChange
ON [dbo].[Students]
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Grade)
    BEGIN
        INSERT INTO StudentGradeAudit (StudentID, OldGrade, NewGrade)
        SELECT
            I.StudentID,
            D.Grade AS OldGrade,
            I.Grade AS NewGrade
        FROM inserted I
        INNER JOIN deleted D ON I.StudentID = D.StudentID;
    END
END;
```

**Trigger Logic Explained:**

- The trigger fires after an UPDATE on the `Students` table.
- It checks if the `Grade` column was updated (`IF UPDATE(Grade)`).
- The trigger uses both `inserted` and `deleted` to capture the old and new grades, inserting this information into the `StudentGradeAudit` table.

**Conclusion:**

The `inserted` and `deleted` tables are powerful tools in T-SQL triggers, providing a means to accurately track and respond to data changes in a database. Understanding their functionality and application is crucial for effective database management and maintaining data integrity, especially in dynamic environments like educational institutions managing student records.

**Quiz 4**

What do the inserted and deleted tables contain?

- The new and old versions of each row that has been inserted or updated
- The new and old versions of each row that has been deleted
- The new version of each row that has been inserted or updated
- The original version of each row that has been deleted or updated

Which DML operation uses the deleted table?

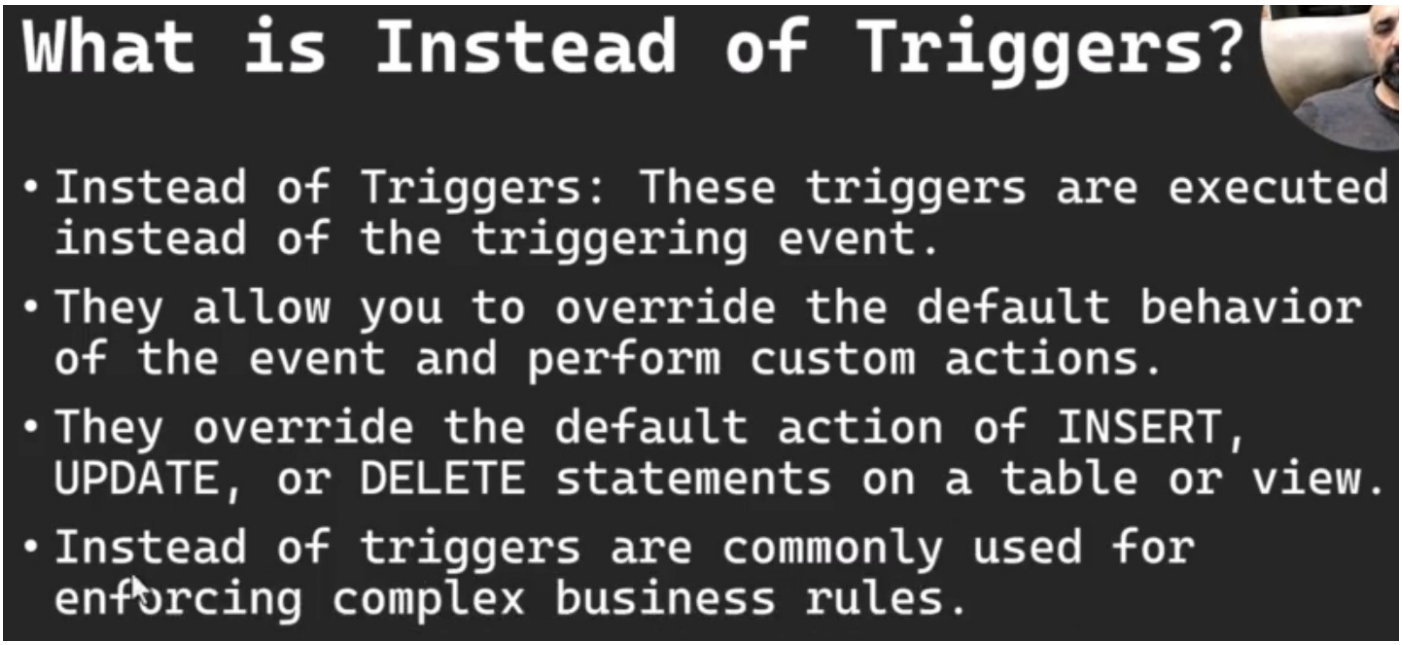
- INSERT
- DELETE
- UPDATE
- None of the above

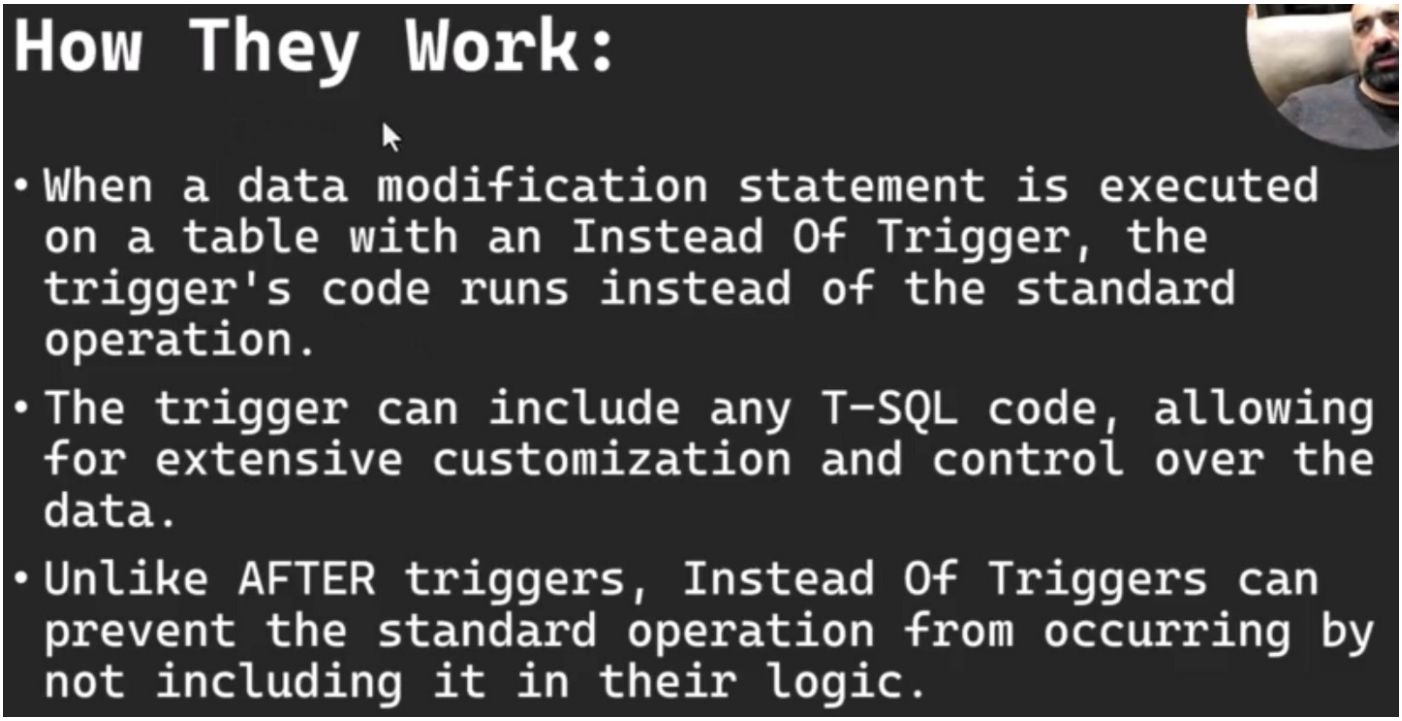
## Understanding Instead Of Triggers in T-SQL

هنا بيقولك انه ال SQL SERVER بيديك القابليه انك تعمل override لل delete statement

وال update وال insert لكن لازم تكون فاهم انت بتعمل ايه

فاانت ممكن مثلا تعمل override لل delete وتكتب كود يروح يعدل علي حاجه معينه في الجدول مثلا بدل مايحذفها






# Understanding Instead Of Triggers in T-SQL

**Introduction**

In this lesson, we will explore "Instead Of Triggers" in T-SQL, a feature of SQL Server. You will learn what they are, how they work, and where they are most effectively used. This lesson is crucial for those looking to add advanced data manipulation and control logic to their SQL Server database interactions.

**What are Instead Of Triggers?**

Instead Of Triggers are a type of database trigger in T-SQL that override the default action of INSERT, UPDATE, or DELETE statements on a table or view. These triggers are essential when you need to perform custom operations or validations that are not achievable with standard SQL constraints or rules.

**Key Characteristics:**

1. Override Default Actions: They replace the standard data modification operation with custom logic defined in the trigger.
2. Versatile Use Cases: Ideal for complex validations, data transformations, and enforcing business rules.
3. Special Tables - `inserted` and `deleted`: These triggers can access the `inserted` and `deleted` tables for context about the data changes.

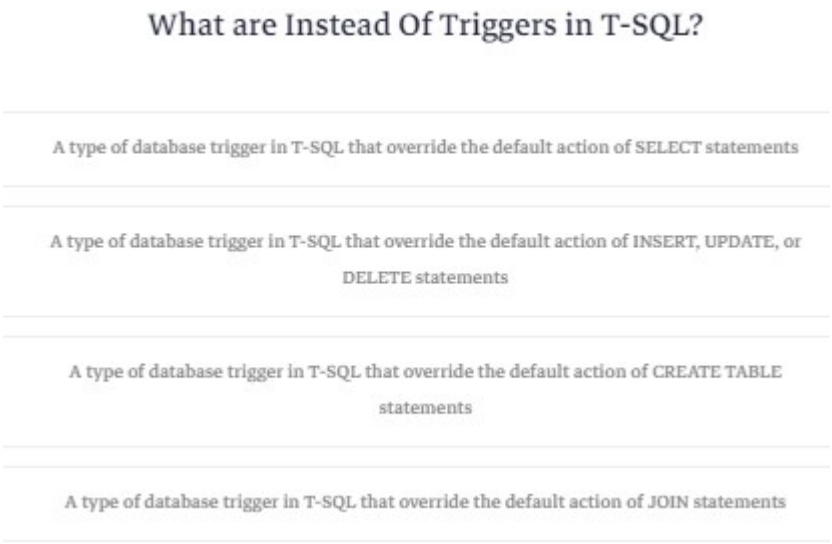
**How They Work:**

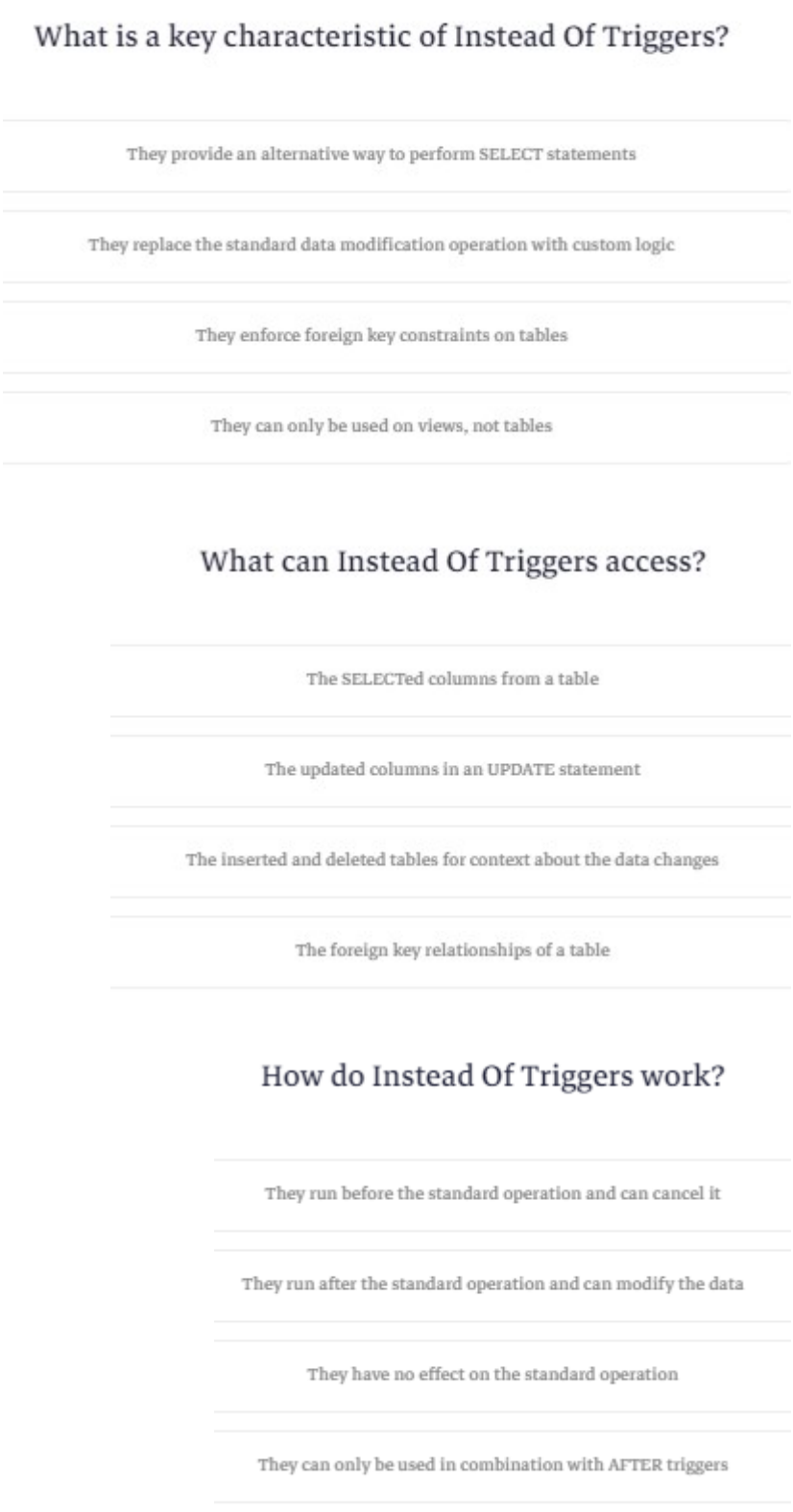
- When a data modification statement is executed on a table with an Instead Of Trigger, the trigger's code runs instead of the standard operation.
- The trigger can include any T-SQL code, allowing for extensive customization and control over the data.
- Unlike AFTER triggers, Instead Of Triggers can prevent the standard operation from occurring by not including it in their logic.

**Typical Use Cases:**

1. Complex Operations on Views: Implementing updateable views based on multiple tables.
2. Data Validation: Enforcing complex validation rules before data changes are committed.
3. Data Transformation: Modifying data before it is inserted or updated to meet specific criteria.
4. Soft Deletes: Marking records as inactive instead of physically deleting them.

## Quiz 1






## "Instead Of Delete" Trigger in T-SQL

ال instead of delete هتعمل override لل delete اللي موجوده في ال sql server

دلوقتي احنا عاوزين نلغي خاصية ال delete من جدول ال students ونستبدلها باننا نخليه active او لا

خلي بالك لو عملت الحوار ده هتغلي ال events بتاعت ال delete وال update اللي كنا عاملينهم لان الكود بتاعهم اصبح غير موجود عندك

تعالي الأول نزود العمود ده

```
alter table students
add IsActive bit default 1

update Students
set IsActive=1

select * from Students
```

طيب تعالي نعمل ال trigger

```
create trigger trg_InsteadOfDeleteStudent on Students
instead of delete
as
begin
update Students
set IsActive=0
from Students inner join deleted
on Students.StudentID=deleted.StudentID
end
```

تعالي نجرب بقي

```
delete Students where StudentID=9

select * from Students
```

## Advanced Scenario for "Instead Of Delete" Trigger in T-SQL

This advanced scenario demonstrates the use of an "Instead Of Delete" trigger in T-SQL for handling soft deletes. In many applications, it's preferable not to permanently remove records for data integrity and historical tracking purposes. Instead, we mark them as inactive or deleted. This technique is known as a soft delete.

**Scenario Setup:**

We'll use the 'Students' table again, but this time, we'll add a new column to indicate whether a record is active. A record marked as inactive will be treated as deleted, without actually being removed from the table.

First, modify the 'Students' table to include a 'IsActive' column:

```
ALTER TABLE Students
ADD IsActive BIT DEFAULT 1;
```

Here, `IsActive` is a bit column where '1' indicates active (not deleted) and '0' indicates inactive (soft deleted).

### Step 2: Creating the Instead Of Delete Trigger

Create an "Instead Of Delete" trigger that, upon a delete operation, will update the 'IsActive' column to '0' instead of physically deleting the row.

```
CREATE TRIGGER trg_InsteadOfDeleteStudent ON Students
INSTEAD OF DELETE
AS
BEGIN
    -- Marking the record as inactive instead of deleting
    UPDATE Students
    SET IsActive = 0
    FROM Students S
    INNER JOIN deleted D ON S.StudentID = D.StudentID;
END;
```

**Explanation:**

- This trigger intercepts DELETE operations on the 'Students' table.
- Instead of deleting, it sets the `IsActive` flag to '0' for the rows that would have been deleted.
- The `deleted` table (used within the trigger) contains the rows that were intended to be deleted.

### Step 3: Testing the Trigger

Test the trigger by attempting to delete a student record and then checking the status of the record.

```
-- Assuming there is a student with StudentID = 1
```

```
-- Attempting to delete a student
DELETE FROM Students WHERE StudentID = 1;


-- Checking the status of the student record
SELECT StudentID, Name, IsActive FROM Students WHERE StudentID = 1;
```

**Expected Result:**

- The record with `StudentID = 1` should still be in the `Students` table.
- The `IsActive` column for this record should now be set to '0', indicating a soft delete.

**Conclusion:**

Using an "Instead Of Delete" trigger for soft deletes is a powerful strategy in scenarios where data preservation is crucial. It allows records to be flagged as inactive without physically removing them from the database, maintaining data integrity and allowing for historical data tracking. This approach is particularly valuable in enterprise applications, auditing, and data warehousing, where maintaining a complete data history is essential.

This advanced use case illustrates how "Instead Of Delete" triggers can be tailored to meet complex business requirements and data management strategies in SQL Server databases.

## Quiz 2

The purpose of using an 'Instead Of Delete' trigger in T-SQL is to override the original delete from SQL and let the trigger handle the action.?

True

False

What does the 'IsActive' column represent in the 'Students' table?

- Whether a student is enrolled in a course
- Whether a student is active in a sports team
- Whether a record is active or inactive
- Whether a record is deleted or not

What does the 'deleted' table represent in the trigger script?

- The records that have been successfully deleted from the table
- The records that were intended to be deleted but were not
- The records that have been inserted into the table
- The records that have been updated in the table

If you have "Instead of Delete" Trigger, the original record will be deleted from the original table.

- True
- False, no it will not be deleted physically.

## Instead of Update Trigger in T-SQL

ال instead of update بتعمل override علي ال update

واحده من استخداماتها انك مثلا عايز تعمل update علي view معين وال view ده الداتا اللي فيه جايه من جدولين هنا ماتقدرش تعمل update عليه غير لما تعمل override لل update

تعالي نعمل الجدولين وال view دول

```
CREATE TABLE PersonalInfo (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Address NVARCHAR(255)
);
```

```sql
CREATE TABLE AcademicInfo (
    StudentID INT PRIMARY KEY,
    Course NVARCHAR(100),
    Grade INT,
    FOREIGN KEY (StudentID) REFERENCES PersonalInfo(StudentID)
);

INSERT INTO PersonalInfo (StudentID, Name, Address) VALUES
(1, 'John Doe', '123 Main St'),
(2, 'Jane Smith', '456 Oak Ave');


INSERT INTO AcademicInfo (StudentID, Course, Grade) VALUES
(1, 'Computer Science', 90),
(2, 'Mathematics', 85);
```

```sql
CREATE VIEW StudentView AS
SELECT P.StudentID, P.Name, P.Address, A.Course, A.Grade
FROM PersonalInfo P
JOIN AcademicInfo A ON P.StudentID = A.StudentID;
```

هنا بقي لما بنعمل override لل update اللي علي ال view احنا في الحقيقه بنعمل update علي كل جدول من الجداول المستخدمه في ال view

```sql
CREATE TRIGGER trg_UpdateStudentView ON StudentView
INSTEAD OF UPDATE
AS
BEGIN
   -- Update PersonalInfo
   UPDATE PersonalInfo
   SET Name = I.Name, Address = I.Address
   FROM PersonalInfo
   INNER JOIN inserted I ON PersonalInfo.StudentID =
I.StudentID;


   -- Update AcademicInfo
   UPDATE AcademicInfo
   SET Course = I.Course, Grade = I.Grade
   FROM AcademicInfo
   INNER JOIN inserted I ON AcademicInfo.StudentID =
I.StudentID;
END;
```

تعالي نجرب كده

```
UPDATE StudentView
SET Name = 'Johnathan Doe', Course = 'Biology', Grade = 92
WHERE StudentID = 1;

SELECT * FROM PersonalInfo WHERE StudentID = 1;
SELECT * FROM AcademicInfo WHERE StudentID = 1;
```

## Updating a Multi-Table View using Instead of Update Trigger

Imagine you have a database with two tables, `PersonalInfo` and `AcademicInfo`, representing different aspects of student data. You then create a view, `StudentView`, which combines these tables to provide a complete profile of each student.

- `PersonalInfo`: Contains columns like `StudentID`, `Name`, and `Address`.
- `AcademicInfo`: Contains columns like `StudentID`, `Course`, and `Grade`.

The view `StudentView` is created with a join on `StudentID` from both tables and includes columns from both `PersonalInfo` and `AcademicInfo`.

**Problem with After Update Trigger**

An "After Update" trigger cannot be used effectively for directly updating this view because:

- The view is based on multiple tables, and a standard UPDATE operation on the view doesn't inherently know how to distribute changes to its underlying tables.
- An "After Update" trigger would only respond after an update attempt on the view, which would fail if the view is not directly updateable.

**Solution with Instead Of Update Trigger**

An "Instead Of Update" trigger on the view can intercept update attempts and appropriately distribute the changes to the underlying tables.

## Step 1: Creating the Tables

Script for `PersonalInfo` Table:

```
CREATE TABLE PersonalInfo (
    StudentID INT PRIMARY KEY,
```

```
    Name NVARCHAR(100),
    Address NVARCHAR(255)
);
```

Script for `AcademicInfo` Table:

```
CREATE TABLE AcademicInfo (
    StudentID INT PRIMARY KEY,
    Course NVARCHAR(100),
    Grade INT,
    FOREIGN KEY (StudentID) REFERENCES PersonalInfo(StudentID)
);
```

## Step 2: Inserting Sample Data

Insert Data into `PersonalInfo`:

```
INSERT INTO PersonalInfo (StudentID, Name, Address) VALUES
(1, 'John Doe', '123 Main St'),
(2, 'Jane Smith', '456 Oak Ave');
```

Insert Data into `AcademicInfo`:

```
INSERT INTO AcademicInfo (StudentID, Course, Grade) VALUES
(1, 'Computer Science', 90),
(2, 'Mathematics', 85);
```

## Step 3: Creating the View

Script for `StudentView`:

```
CREATE VIEW StudentView AS
SELECT P.StudentID, P.Name, P.Address, A.Course, A.Grade
FROM PersonalInfo P
JOIN AcademicInfo A ON P.StudentID = A.StudentID;
```

## Step 4: Creating the Instead Of Update Trigger

Script for the Trigger on `StudentView`:

```
CREATE TRIGGER trg_UpdateStudentView ON StudentView
INSTEAD OF UPDATE
AS
BEGIN
    -- Update PersonalInfo
    UPDATE PersonalInfo
    SET Name = I.Name, Address = I.Address
    FROM PersonalInfo
    INNER JOIN inserted I ON PersonalInfo.StudentID = I.StudentID;


    -- Update AcademicInfo
    UPDATE AcademicInfo
    SET Course = I.Course, Grade = I.Grade
    FROM AcademicInfo
    INNER JOIN inserted I ON AcademicInfo.StudentID = I.StudentID;
END;
```

## Step 5: Testing the Setup

To test this setup, you can attempt to update the `StudentView` and then check both the `PersonalInfo` and `AcademicInfo` tables to see if the updates were applied correctly.

Update Statement for Testing:

```
UPDATE StudentView
SET Name = 'Johnathan Doe', Course = 'Biology', Grade = 92
WHERE StudentID = 1;
```

Check the Updates:

```
SELECT * FROM PersonalInfo WHERE StudentID = 1;
SELECT * FROM AcademicInfo WHERE StudentID = 1;
```

This set of scripts will create the required tables, view, and trigger for the example scenario. By running these scripts in your SQL Server environment, you can see how the "Instead Of Update" trigger works to update a view based on multiple underlying tables.

Creating the Instead Of Update Trigger:

```
CREATE TRIGGER trg_UpdateStudentView ON StudentView
INSTEAD OF UPDATE
AS
BEGIN
    -- Update PersonalInfo
    UPDATE PersonalInfo
    SET Name = I.Name, Address = I.Address
    FROM PersonalInfo
    INNER JOIN inserted I ON PersonalInfo.StudentID = I.StudentID;


    -- Update AcademicInfo
    UPDATE AcademicInfo
    SET Course = I.Course, Grade = I.Grade
    FROM AcademicInfo
    INNER JOIN inserted I ON AcademicInfo.StudentID = I.StudentID;
END;
```

**In this trigger:**

- When an UPDATE statement is issued on `StudentView`, the trigger activates.
- Instead of performing a standard update (which isn't directly feasible on the view), it updates the corresponding rows in the `PersonalInfo` and `AcademicInfo` tables based on the data in the `inserted` pseudo-table.
- This allows the view to be "updateable" in a logical sense, even though it's based on multiple underlying tables.

## Conclusion

This scenario illustrates a case where an "Instead Of Update" trigger is necessary and cannot be replaced by an "After Update" trigger. It's a powerful example of the flexibility and utility of "Instead Of Update" triggers in handling complex data relationships and views in SQL Server.

## Quiz 3

What is the purpose of the StudentView?

To combine unrelated data from different tables

To provide a complete profile of each student

To store personal information of students

Why can't an 'After Update' trigger be used effectively for updating the StudentView?

The StudentView is not directly updateable

The StudentView is based on multiple tables

Both of the above


## Instead of Insert Trigger in T-SQL

هنعمل ال insert trigger علي ال view اللي عملناه بحيث نعمل insert علي الجدولين اللي بيتكون منهم ال view

ده ال trigger

```
create trigger trg_InsertStudentView on StudentView
instead of insert
as
begin
insert into PersonalInfo
select StudentID,Name,Address from inserted

insert into AcademicInfo
select StudentID,Course,Grade from inserted

end
```

تعالي نجرب

```
INSERT INTO StudentView (StudentID, Name, Address, Course, Grade)
VALUES (3, 'Alice Johnson', '789 Pine Rd', 'Physics', 88);

select * from StudentView

SELECT * FROM PersonalInfo WHERE StudentID = 3;
SELECT * FROM AcademicInfo WHERE StudentID = 3;
```

## Implementing Instead Of Insert Triggers in T-SQL for Multi-Table Views

In this lesson, we'll explore how to use "Instead Of Insert" triggers in T-SQL, specifically in the context of inserting data into views that are based on multiple underlying tables. This is a common scenario in complex databases where views simplify user interaction but complicate direct data insertion.

Understand and implement an Instead Of Insert trigger for a view that combines multiple tables. By the end of this lesson, you'll be able to create such a trigger to facilitate data insertion into complex views.

### What are Instead Of Insert Triggers?

Instead Of Insert triggers are a special type of trigger in T-SQL that intercept and replace the standard INSERT operation on a table or view. They are particularly useful when you need to insert data into a view that spans multiple underlying tables.

### Scenario Setup:

Consider a database with two related tables, `PersonalInfo` and `AcademicInfo`, and a combined view, `StudentView`. The challenge is to allow insertions into this view despite it being based on two separate tables.

- `PersonalInfo`: Contains personal student information like name and address.
- `AcademicInfo`: Contains academic details like course and grade.

### Step 1: Creating the Tables and View

**PersonalInfo Table:**

```
CREATE TABLE PersonalInfo (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Address NVARCHAR(255)
);
```

**AcademicInfo Table:**

```
CREATE TABLE AcademicInfo (
    StudentID INT PRIMARY KEY,
    Course NVARCHAR(100),
    Grade INT,
    FOREIGN KEY (StudentID) REFERENCES PersonalInfo(StudentID)
);
```

**StudentView:**

```
CREATE VIEW StudentView AS
SELECT P.StudentID, P.Name, P.Address, A.Course, A.Grade
FROM PersonalInfo P
JOIN AcademicInfo A ON P.StudentID = A.StudentID;
```

### Step 2: Implementing the Instead Of Insert Trigger

**Trigger on StudentView:**

```
CREATE TRIGGER trg_InsertStudentView ON StudentView
INSTEAD OF INSERT
AS
BEGIN
    -- Insert into PersonalInfo
    INSERT INTO PersonalInfo (StudentID, Name, Address)
    SELECT StudentID, Name, Address FROM inserted;

    -- Insert into AcademicInfo
    INSERT INTO AcademicInfo (StudentID, Course, Grade)
    SELECT StudentID, Course, Grade FROM inserted;
END;
```

**In this trigger:**

- When an INSERT statement is executed on `StudentView`, the trigger activates.
- The trigger splits the incoming data, inserting relevant portions into `PersonalInfo` and `AcademicInfo`.

### Step 3: Testing the Trigger

To ensure the trigger works correctly, insert a record into `StudentView` and verify the data in both underlying tables.

**Insert Statement for Testing:**

```
INSERT INTO StudentView (StudentID, Name, Address, Course, Grade)
VALUES (3, 'Alice Johnson', '789 Pine Rd', 'Physics', 88);
```

**Verify the Insertions:**

```
SELECT * FROM PersonalInfo WHERE StudentID = 3;
SELECT * FROM AcademicInfo WHERE StudentID = 3;
```

**Conclusion:**

"Instead Of Insert" triggers are invaluable in scenarios involving complex views over multiple tables, especially when direct data insertion is not straightforward. By learning to implement these triggers, you can handle data insertion into complex views efficiently, maintaining data integrity and consistency across your SQL Server database.

## Quiz 4

What is the purpose of 'Instead Of Insert' triggers in T-SQL?

- To intercept and replace the standard INSERT operation on a table or view
- To delete data from a table or view
- To update data in a table or view

When are 'Instead Of Insert' triggers particularly useful?

- When you need to insert data into a view that spans multiple underlying tables
- When you need to delete data from a view
- When you need to update data in a view

What is the purpose of the 'StudentView' view in the given scenario?

- To combine data from the PersonalInfo and AcademicInfo tables
- To delete data from the PersonalInfo and AcademicInfo tables
- To update data in the PersonalInfo and AcademicInfo tables

What does the 'INSERT INTO StudentView' statement in the 'Testing the Trigger' section do?

- Inserts a record into the StudentView view
- Deletes a record from the StudentView view
- Updates a record in the StudentView view

How does the 'Instead Of Insert' trigger split the incoming data?

- By inserting relevant portions into PersonalInfo and AcademicInfo
- By deleting relevant portions from PersonalInfo and AcademicInfo
- By updating relevant portions in PersonalInfo and AcademicInfo

## Introduction to Cursors in T-SQL

هنبدأ نتعلم عن ال cursor وأول حاجه عايزك تتعلمها انك ماتستخدمهوش غير لما تتزنق اوي لانه بيبطأ الداتابيز

ال cursor هوا object بيخليك تلف عالداتا وفيه منه اربع أنواع

1- static cursor :- بياخد نسخه من الداتا او سكرين شوت ويعرضهالك تعمل عليها اللي انت عايزه بس لو حصل تعديل عالداتا مش هتسمع في النسخه بتاعتك اللي ظهرتلك

2- Dynamic cursor :- بياخد سكرين شوت اه بس بيفتح connection مع الداتا بحيث لو حصل تعديل يسمع عندك

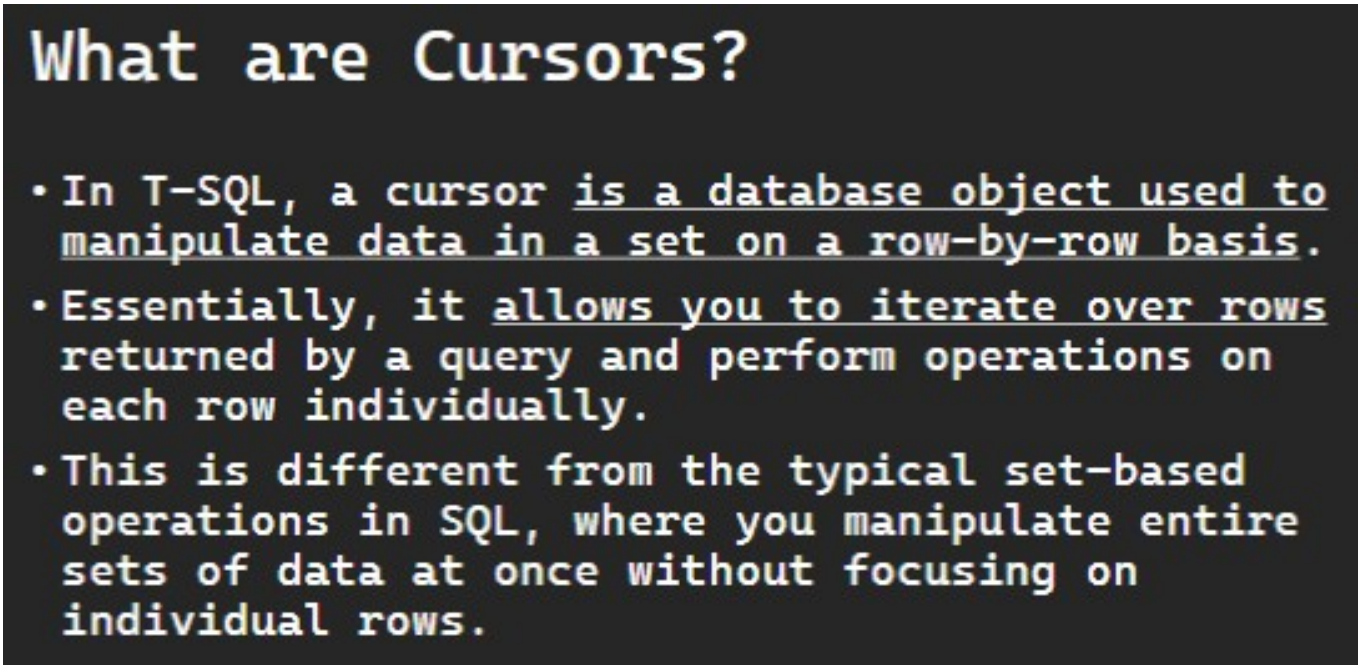
3- Forward only cursor :- زيه زي ال data reader بيخليك تمشي في اتجاه واحد من فوق لتحت

4- Scrollable cursor :- بيخليك تمشي من فوق لتحت او من تحت لفوق

الفكره انك لما بتتعامل مع الداتا بيز فاانت بتتعامل مع ال data set كلها بس لو عايز تتعامل معاها كل صف علي حدي زي انك تمسك كل شخص في الداتابيز وتبعتله ايميل مثلا بتستخدم ال cursor ومعظم الحالات بتتحل بدون استخدام ال cursor

كلمة set based معناها انك بتتعامل مع الجدول علي انه وحده واحده

ولازم بعد ماتفتح ال cursor تقفله



# Why Use Cursors?

While SQL is inherently designed for set-based operations, there are scenarios where you might need to work with data one row at a time. This is where cursors come into play. They are particularly useful when:

1. Sequential Processing is Required: You need to process data in a specific order, one row at a time.
2. Complex Logic per Row: Each row requires complex processing or decision-making that cannot be easily or efficiently expressed in a set-based approach.
3. Interactivity: Situations where the data needs to be processed interactively, such as in applications that allow users to scroll through individual rows.

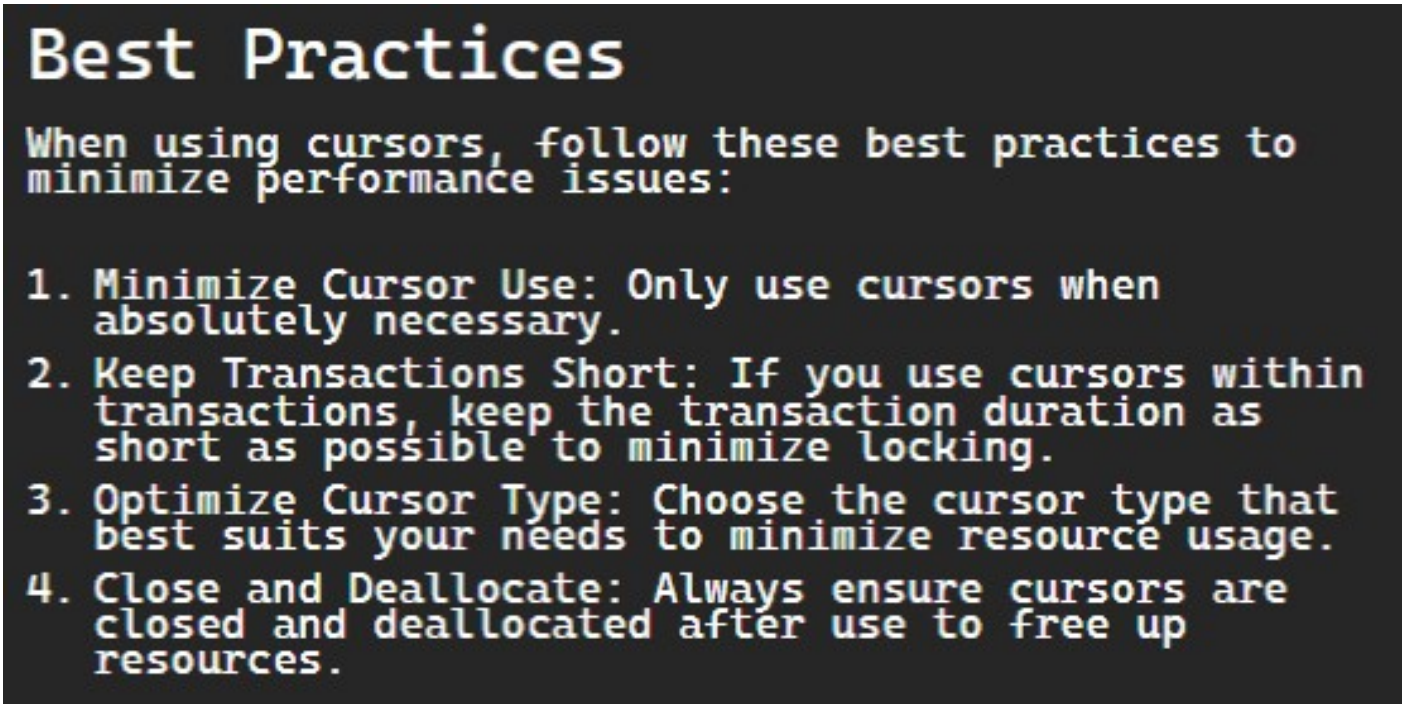
# Types of Cursors

1. Static Cursors: These create a snapshot of the data when the cursor is opened. Changes made to the data in the database after the cursor is opened are not reflected in the cursor.
2. Dynamic Cursors: These reflect changes made to the data in the database while the cursor is open.
3. Forward-Only Cursors: As the name suggests, these cursors can only move forward through the data.
4. Scrollable Cursors: These allow movement both forward and backward through the data and can jump to specific rows.

# Performance Considerations

Cursors can be resource-intensive and potentially lead to performance issues, particularly in high-volume databases. They should be used judiciously, and it's often recommended to explore set-based alternatives before resorting to cursors. Some key considerations include:

1. Overhead: Cursors can involve significant overhead, especially when dealing with large datasets.
2. Locking and Concurrency: Using cursors can lead to extended locking of rows or tables, potentially affecting concurrency.
3. Alternatives: Often, tasks requiring cursors can be restructured into set-based operations, which are typically more efficient in SQL.




## Introduction to Cursors in T-SQL

**What are Cursors?**

In T-SQL, a cursor is a database object used to manipulate data in a set on a row-by-row basis. Essentially, it allows you to iterate over rows returned by a query and perform operations on each row individually. This is different from the typical set-based operations in SQL, where you manipulate entire sets of data at once without focusing on individual rows.

**Why Use Cursors?**

While SQL is inherently designed for set-based operations, there are scenarios where you might need to work with data one row at a time. This is where cursors come into play. They are particularly useful when:

1. Sequential Processing is Required: You need to process data in a specific order, one row at a time.
2. Complex Logic per Row: Each row requires complex processing or decision-making that cannot be easily or efficiently expressed in a set-based approach.
3. Interactivity: Situations where the data needs to be processed interactively, such as in applications that allow users to scroll through individual rows.

**Types of Cursors**

T-SQL supports several types of cursors, each suited to different scenarios. The main types include:

1. **Static Cursors:** These create a snapshot of the data when the cursor is opened. Changes made to the data in the database after the cursor is opened are not reflected in the cursor.
2. **Dynamic Cursors:** These reflect changes made to the data in the database while the cursor is open.
3. **Forward-Only Cursors:** As the name suggests, these cursors can only move forward through the data.
4. **Scrollable Cursors:** These allow movement both forward and backward through the data and can jump to specific rows.

**Performance Considerations**

Cursors can be resource-intensive and potentially lead to performance issues, particularly in high-volume databases. They should be used judiciously, and it's often recommended to explore set-based alternatives before resorting to cursors. Some key considerations include:

- Overhead: Cursors can involve significant overhead, especially when dealing with large datasets.
- Locking and Concurrency: Using cursors can lead to extended locking of rows or tables, potentially affecting concurrency.
- Alternatives: Often, tasks requiring cursors can be restructured into set-based operations, which are typically more efficient in SQL.

**Best Practices**

When using cursors, follow these best practices to minimize performance issues:

1. Minimize Cursor Use: Only use cursors when absolutely necessary.
2. Keep Transactions Short: If you use cursors within transactions, keep the transaction duration as short as possible to minimize locking.
3. Optimize Cursor Type: Choose the cursor type that best suits your needs to minimize resource usage.
4. Close and Deallocate: Always ensure cursors are closed and deallocated after use to free up resources.

**Conclusion**

Cursors in T-SQL offer a way to perform row-by-row data manipulation, bridging the gap between set-based SQL operations and procedural programming. While powerful, they should be used sparingly and with an understanding of their impact on database performance. For many scenarios, set-based solutions are preferable and should be considered first. Cursors, however, remain a valuable tool in the SQL developer's toolkit for those specific cases where set-based operations fall short.

## Quiz 1

### What is a cursor in T-SQL?

- A database object used to manipulate data in a set on a row-by-row basis.
- A function that allows you to perform calculations on data sets.
- A data type used to store multiple values in a single variable.
- A command used to create and modify database tables.

### When is it appropriate to use cursors in T-SQL?

- When you want to perform set-based operations.
- When you need to process data in a specific order, one row at a time.
- When you want to manipulate entire sets of data efficiently.
- When you want to avoid complex logic per row.

What types of cursors does T-SQL support?

- Static Cursors
- Dynamic Cursors
- Forward-Only Cursors
- Scrollable Cursors
- All of the above

What are some performance considerations when using cursors in T-SQL?

- Cursors have no impact on database performance.
- Cursors can be resource-intensive and potentially lead to performance issues.
- Cursors are always more efficient than set-based operations.
- Cursors do not require any locking or concurrency management.

What are some best practices when using cursors in T-SQL?

- Use cursors for all data manipulation tasks.
- Keep transactions with cursors as long as possible for better performance.
- Choose the cursor type that uses the least amount of resources.
- Always ensure cursors are closed and deallocated after use.

## Static Cursors in T-SQL

هناخد ال static cursor وهوا بياخد نسخه من الداتا بيوريهالك ولو حصل تعديل عالداتا الاصليه مش هتظهر عندك

هوا بيستهلك memory بس مش بيعمل lock للداتابيز لانه بعد مابيجيبلك الداتا بيفصل الاتصال بينه وبين الداتا بيز

بينفع لما ماتكونش مهتم بالتغييرات اللي بتحصل عالداتا الاصليه او انك عايز تعمل تقارير او تحلل الداتا

من مميزاته انه الداتا بتكون ثابته وانك مش بتعمل lock عالداتابيز

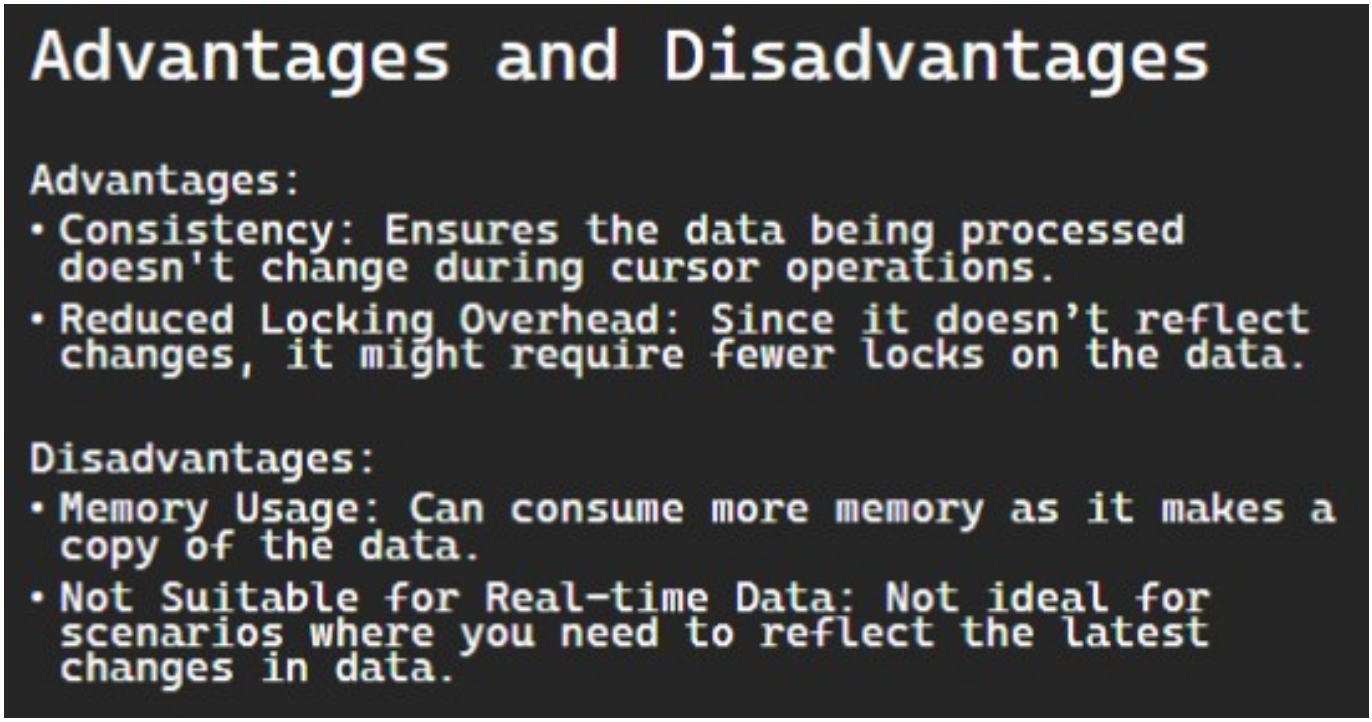
عيوبه انه ممكن يستهلك memory كتير لو الداتا اللي بينسخها حجمها كبير وماينفعش تستخدمه في حاجه بتحتاج مراقبه عالتغييرات اللي بتتم عالداتا زي الاسهم بتاعت البورصه كده

## What is Static Cursors?

- Static cursors in T-SQL create a fixed snapshot of the data at the time the cursor is opened.
- This means any changes made to the underlying data after the cursor has been opened will not be reflected in the cursor's result set.
- Static cursors are particularly useful when you need a consistent set of data throughout the cursor's life and do not need to track real-time changes in the underlying data.

## When to Use Static Cursors?

- Report Generation: When generating reports where data consistency is essential and should not reflect changes during the report generation process.
- Data Analysis: For analysis tasks where you need to work on a stable dataset.
- Not Real-Time.




طريقة تعريفه زي طريقة تعريف أي variable بس بتكتب بعديه for وبعدها ال query اللي هتجيبلك الداتا عشان تخزنها في ال cursor

```
declare static_cursor cursor static for
select Students.StudentID , Students.Name,Students.Grade
from Students
```

عشان تبدأ تشتغل علي ال cursor اللي عملته ده لازم الأول تفتحه

وبعدين بتعمل متغيرات تخزن فيها الداتا اللي في كل record

```
open static_Cursor

declare @StudentID INT,@Name nvarchar(50) , @Grade int
```

طيب احنا دلوقتي عايزين نلف علي كل صف ونجيب الداتا اللي فيه ونحطها في المتغيرات اللي عملناها

عشان كده هنستخدم fetch next في اننا نشاور علي اول صف في الجدول

بتكتب fetch next وبعدين from وبعديه اسم ال cursor اللي فيه الداتا وبعدين into وبعدين اسامي المتغيرات اللي هنخزن فيها الداتا

```
fetch next from static_cursor into @StudentID,@Name,@Grade
```

بعدين هنستخدم ال while loop عشان نلف عالداتا صف صف

طيب الشرط هيكون ايه؟

الشرط هيكون انه ال fetch next يكون فيه داتا

هنعرفها ازاي؟

عن طريق متغير اسمه @fetch_status ده لو قيمته بصفر يبقي ال fetch next في داتا

يعني طول مافيه record هتفضل ال loop شغاله

ونكتب الكود بتاعنا اللي عايزين نعمله وبعدين نحرك ال fetch next للصف اللي بعديه

```
while @@FETCH_STATUS=0
begin
print 'Student Name :' + @Name + 'Grade :' + cast(@Grade as
nvarchar(10))
fetch next from static_cursor into @StudentID,@Name,@Grade
end
```

وطبعا ماتنساش تقفل ال cursor وتعمله deallocate بعد ماتخلص

```
close static_cursor
deallocate static_cursor
```

وده الكود كله

```
declare static_cursor cursor static for
select Students.StudentID , Students.Name,Students.Grade
from Students

open static_Cursor

declare @StudentID INT,@Name nvarchar(50) , @Grade int

fetch next from static_cursor into @StudentID,@Name,@Grade

while @@FETCH_STATUS=0
begin
print 'Student Name :' + @Name + 'Grade :' + cast(@Grade as nvarchar(10))
fetch next from static_cursor into @StudentID,@Name,@Grade
end

close static_cursor
deallocate static_cursor
```

## Understanding Static Cursors in T-SQL

**Introduction to Static Cursors**

Static cursors in T-SQL create a fixed snapshot of the data at the time the cursor is opened. This means any changes made to the underlying data after the cursor has been opened will not be reflected in the cursor's result set. Static cursors are particularly useful when you need a consistent

set of data throughout the cursor's life and do not need to track real-time changes in the underlying data.

**When to Use Static Cursors**

- Report Generation: When generating reports where data consistency is essential and should not reflect changes during the report generation process.
- Data Analysis: For analysis tasks where you need to work on a stable dataset.

**Advantages and Disadvantages**

- Advantages:
  - Consistency: Ensures the data being processed doesn't change during cursor operations.
  - Reduced Locking Overhead: Since it doesn’t reflect changes, it might require fewer locks on the data.
- Disadvantages:
  - Memory Usage: Can consume more memory as it makes a copy of the data.
  - Not Suitable for Real-time Data: Not ideal for scenarios where you need to reflect the latest changes in data.

**Conclusion**

Static cursors are a powerful tool in T-SQL, offering a stable and consistent view of data. They are ideal in scenarios where data consistency throughout the cursor operation is paramount. However, their use should be judicious, considering their impact on memory and the requirement for up-to-date data. Always evaluate if a set-based operation can be used before opting for cursors.

**Quiz 2**

## What is the purpose of static cursors in T-SQL?

- To create a fixed snapshot of the data at the time the cursor is opened
- To track real-time changes in the underlying data
- To consume more memory
- To reflect the latest changes in data

## When are static cursors particularly useful?

- When you need to track real-time changes in the underlying data
- When you need a consistent set of data throughout the cursor's life
- When memory usage is not a concern
- When you need to reflect the latest changes in data

## What is an advantage of static cursors?

- Ensures the data being processed doesn't change during cursor operations
- Requires fewer locks on the data
- Consumes less memory
- Reflects the latest changes in data

## What is a disadvantage of static cursors?

- Reflects the latest changes in data
- Requires more memory usage
- Tracks real-time changes in the underlying data
- Doesn't require any locks on the data

When should a set-based operation be considered instead of using cursors?

- Never
- Always
- Only when dealing with large datasets
- Always evaluate if a set-based operation can be used before opting for cursors

## Dynamic Cursors in T-SQL

ال dynamic cursor هيا ابطأ من ال static cursor لانه بيخلي ال connection مع الداتابيز شغال عشان يخلي الداتا بتاعتك متحدثة اول بأول

مميزاته انك بتشوف الداتا متحدثه اول بأول

عيوبه انه بيعمل lock عالداتا والتحكم فيه اصعب من ال static

لو انت شغال في ال loop وفيه record انضاف هينضاف عندك

وده مثال عليه نفس الشغل بتاع ال static

```
declare dynamic_cursor cursor dynamic for
select Students.StudentID , Students.Name,Students.Grade
from Students

open dynamic_cursor

declare @StudentID INT,@Name nvarchar(50) , @Subject nvarchar(50)

fetch next from dynamic_cursor into @StudentID,@Name,@Subject

while @@FETCH_STATUS=0
begin
print 'Student Name :' + @Name + 'Grade :' + cast(@Subject as nvarchar(10))
fetch next from dynamic_cursor into @StudentID,@Name,@Subject
end

close dynamic_cursor
deallocate dynamic_cursor
```

# What is Dynamic Cursors?

- Dynamic cursors are a type of cursor in T-SQL that provide a live view of the database.
- Unlike static cursors, dynamic cursors reflect changes made to the underlying data while the cursor is open.
- This means if a row is updated, inserted, or deleted in the table, these changes will be visible in the cursor's result set.

# When to Use Dynamic Cursors?

- Real-time Data Processing: Ideal for tasks that require up-to-date information from the database.
- Monitoring Changes: Useful in scenarios where you need to monitor and react to data changes.

# Advantages and Disadvantages

Advantages:

- Real-time Data: Reflects the latest data changes in the database.
- Flexibility: Allows for more dynamic and responsive data processing.

Disadvantages:

- Performance Overhead: Can be slower and more resource-intensive due to tracking changes.
- Complexity: Managing a dynamic cursor can be more complex compared to static cursors.

# Understanding Dynamic Cursors in T-SQL

### Introduction to Dynamic Cursors

Dynamic cursors are a type of cursor in T-SQL that provide a live view of the database. Unlike static cursors, dynamic cursors reflect changes made to the underlying data while the cursor is open. This means if a row is updated, inserted, or deleted in the table, these changes will be visible in the cursor's result set.

### When to Use Dynamic Cursors

- Real-time Data Processing: Ideal for tasks that require up-to-date information from the database.
- Monitoring Changes: Useful in scenarios where you need to monitor and react to data changes.

### Advantages and Disadvantages

- Advantages:
  - Real-time Data: Reflects the latest data changes in the database.
  - Flexibility: Allows for more dynamic and responsive data processing.
- Disadvantages:
  - Performance Overhead: Can be slower and more resource-intensive due to tracking changes.
  - Complexity: Managing a dynamic cursor can be more complex compared to static cursors.

### Conclusion

Dynamic cursors are a powerful feature in T-SQL for scenarios requiring up-to-date data from a database. They are particularly useful for real-time data processing and monitoring. However, they should be used judiciously due to their potential performance overhead and complexity. Always consider if there are more efficient ways to achieve the same result, such as using set-based operations, before opting for a dynamic cursor.

**Quiz 3**

## What is the purpose of dynamic cursors in T-SQL?

- To provide a live view of the database.
- To improve performance of queries.
- To simplify database management.

## How do dynamic cursors differ from static cursors?

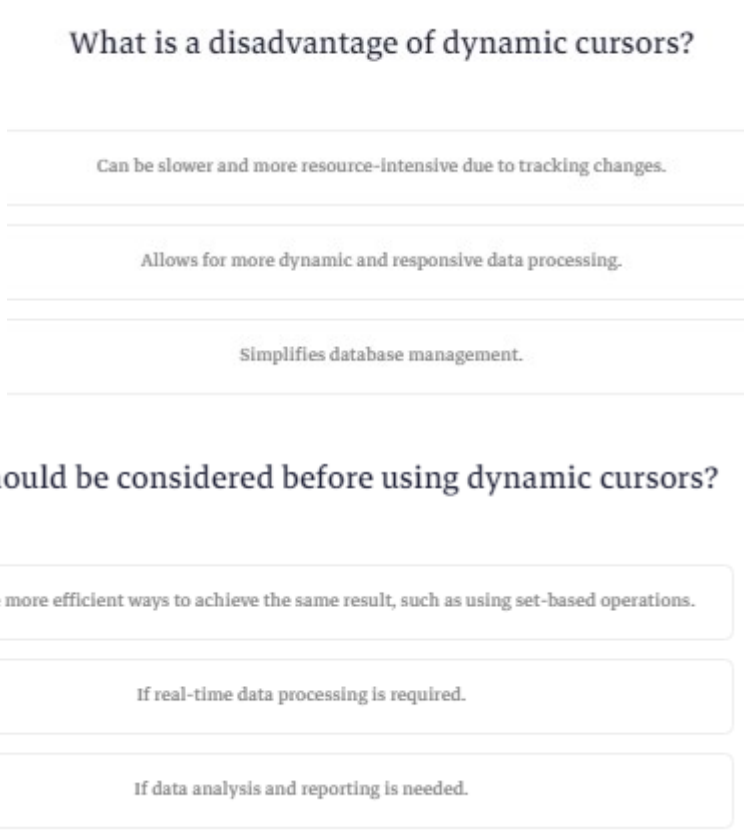
- Dynamic cursors reflect changes made to the underlying data while the cursor is open.
- Static cursors are faster and more resource-intensive.
- Static cursors allow for more dynamic and responsive data processing.

## In what scenarios are dynamic cursors particularly useful?

- Real-time data processing and monitoring changes.
- Batch processing of large data sets.
- Data analysis and reporting.

## What is an advantage of dynamic cursors?

- Reflects the latest data changes in the database.
- Improves query performance.
- Simplifies data management.




## Forward-Only Cursors in T-SQL

هنا بيقولك انك ممكن تفتح ال static اوال dynamic cursor بحيث يكون forward only اللي هوا ال record اللي يتخطاه مايرجعلوش تاني وده الأسرع

بتستخدمه في الحاجات البسيطه او لما تكون عارف انك هتمشي عالداتا في اتجاه واحد وعايز العمليه تكون اسرع

طريقته نفسها طريقة ال static cursor او ال dynamic بس بتزود كلمة forward_only

تقدر ماتكتبش ان كان static او dynamic وال default هياخده من ال configuration بتاع ال sql server عشان كده يفضل انك تكتبها

```
declare forward_only_static_cursor cursor static forward_only for
select Students.StudentID , Students.Name,Students.IsActive
from Students

open forward_only_static_cursor

declare @StudentID INT,@Name nvarchar(50) , @IsActive bit

fetch next from forward_only_static_cursor into @StudentID,@Name,@IsActive

while @@FETCH_STATUS=0
begin
print 'Student Name :' + @Name + 'Grade :' + cast(@IsActive as nvarchar(10))
fetch next from forward_only_static_cursor into @StudentID,@Name,@IsActive
end

close forward_only_static_cursor
deallocate forward_only_static_cursor
```

## What is Forward-Only Cursors Cursors?

- Forward-Only Cursors are the simplest type of cursors in SQL Server.
- As the name suggests, they can only move forward through the result set.
- This type of cursor does not support backward movement or scrolling to specific rows in the result set.
- Their simplicity often translates to better performance compared to more complex cursor types.

## When to Use Forward-Only Cursors?

- Simple Data Retrieval: Best for scenarios where you only need a simple, sequential read through a dataset.
- Performance Consideration: They are generally faster than other cursor types due to their simplicity.

## Advantages and Disadvantages

Advantages:

- Performance: Typically faster than other cursor types due to reduced overhead.
- Simplicity: Easier to understand and use, especially for basic data retrieval tasks.

Disadvantages:

- Limited Flexibility: Cannot move backward or jump to specific rows.
- Not Suitable for Complex Operations: Less suited for tasks requiring advanced cursor functionalities.

# Understanding Forward-Only Cursors in T-SQL

## Introduction to Forward-Only Cursors

Forward-Only Cursors are the simplest type of cursors in SQL Server. As the name suggests, they can only move forward through the result set. This type of cursor does not support backward movement or scrolling to specific rows in the result set. Their simplicity often translates to better performance compared to more complex cursor types.

## When to Use Forward-Only Cursors

- Simple Data Retrieval: Best for scenarios where you only need a simple, sequential read through a dataset.
- Performance Consideration: They are generally faster than other cursor types due to their simplicity.

## Advantages and Disadvantages

- Advantages:
    - Performance: Typically faster than other cursor types due to reduced overhead.
    - Simplicity: Easier to understand and use, especially for basic data retrieval tasks.
- Disadvantages:
    - Limited Flexibility: Cannot move backward or jump to specific rows.
    - Not Suitable for Complex Operations: Less suited for tasks requiring advanced cursor functionalities.

## Conclusion

Forward-Only Cursors in T-SQL are a great tool for simple, sequential data access patterns. They offer performance benefits due to their simplicity but come with limitations in terms of flexibility. Understanding when and how to use them effectively is key to leveraging their strengths while avoiding scenarios where more complex cursors would be more appropriate. As always, consider whether a set-based approach could be more efficient before opting for any cursor.

# Quiz 4

## What are Forward-Only Cursors in SQL Server?

- Cursors that can only move forward through the result set.
- Cursors that can move backward or jump to specific rows.
- Cursors that support scrolling to specific rows but not backward movement.
- Cursors that can only move backward through the result set.

## Which of the following is an advantage of Forward-Only Cursors?

- Performance: Typically faster than other cursor types due to reduced overhead.
- Flexibility: Can move backward or jump to specific rows.
- Simplicity: Easier to understand and use, especially for basic data retrieval tasks.
- Suitable for complex operations requiring advanced cursor functionalities.

## What is a disadvantage of Forward-Only Cursors?

- Limited Flexibility: Cannot move backward or jump to specific rows.
- Performance: Slower than other cursor types due to increased overhead.
- Complexity: Difficult to understand and use, especially for basic data retrieval tasks.
- Suitable for complex operations requiring advanced cursor functionalities.

When are Forward-Only Cursors generally faster than other cursor types?

- For simple, sequential read through a dataset.
- For tasks requiring advanced cursor functionalities.
- For scrolling to specific rows in the result set.
- For scenario where moving backward is required.

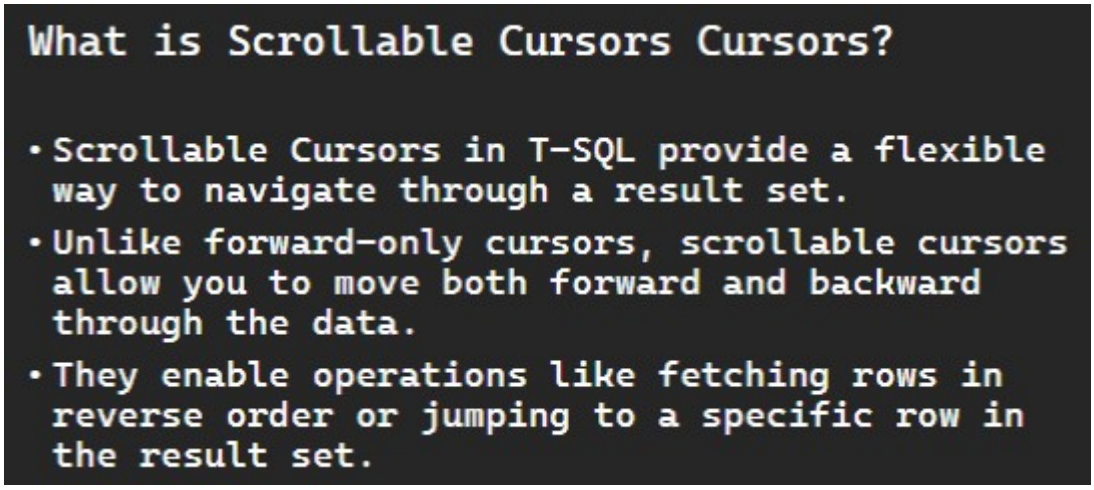
## Scrollable Cursors in T-SQL

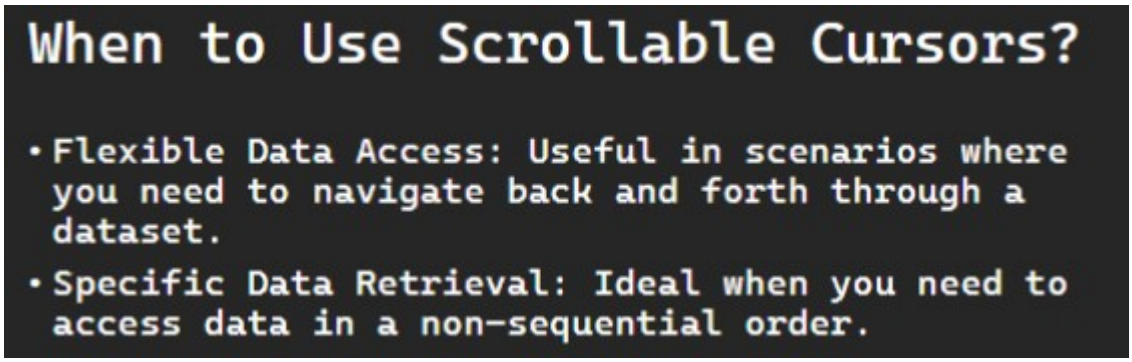
ال scrollable cursor بتقدر تنتقل للامام والخلف في الداتا وهوا ابطأ من ال forward only وتقدر تستخدمه مع ال static والـ dynamic

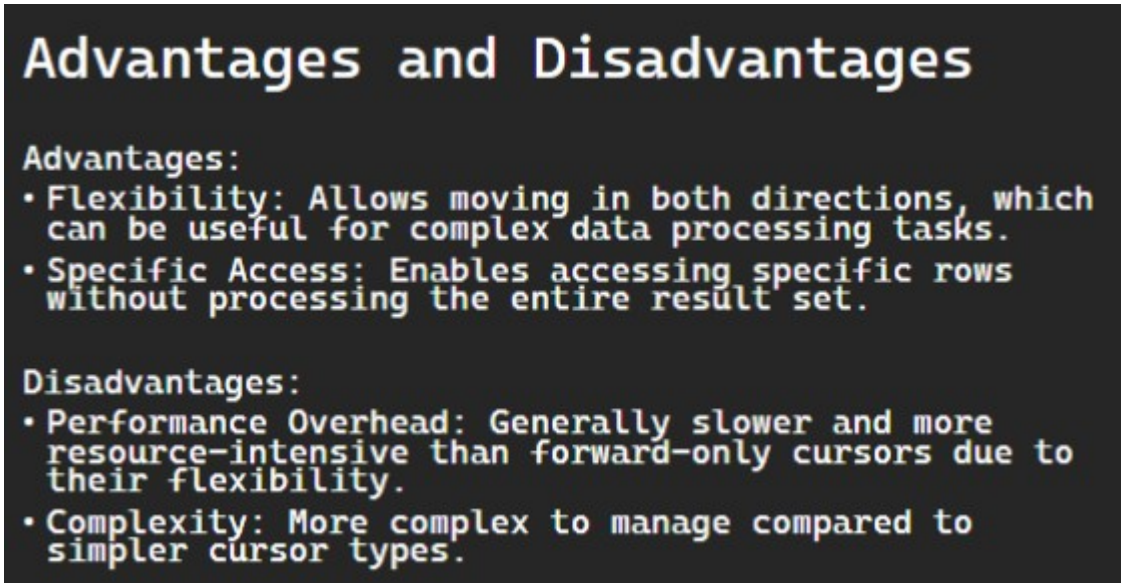
نفس الشغل بس هنا بيزيد عليه fetch previous عشان تتحرك backward

وطريقته بتكتب scroll بدل forward only

```
use C21_DB1;

DECLARE scroll_cursor CURSOR STATIC SCROLL FOR
SELECT StudentID, Name, Grade FROM dbo.Students;

OPEN scroll_cursor;

DECLARE @StudentID int, @Name nvarchar(50), @Grade int;

FETCH NEXT FROM scroll_cursor INTO @StudentID, @Name, @Grade;

WHILE @@FETCH_STATUS = 0
BEGIN

    PRINT 'Student Name: ' + @Name + ', Grade: ' + CAST(@Grade AS NVARCHAR(10));

 --FETCH PRIOR FROM scroll_cursor INTO @StudentID, @Name, @Grade;

    FETCH NEXT FROM scroll_cursor INTO @StudentID, @Name, @Grade;
END

CLOSE scroll_cursor;

DEALLOCATE scroll_cursor;
```




# Understanding Scrollable Cursors in T-SQL

**Introduction to Scrollable Cursors**

Scrollable Cursors in T-SQL provide a flexible way to navigate through a result set. Unlike forward-only cursors, scrollable cursors allow you to move both forward and backward through the data. They enable operations like fetching rows in reverse order or jumping to a specific row in the result set.

**When to Use Scrollable Cursors**

- Flexible Data Access: Useful in scenarios where you need to navigate back and forth through a dataset.
- Specific Data Retrieval: Ideal when you need to access data in a non-sequential order.

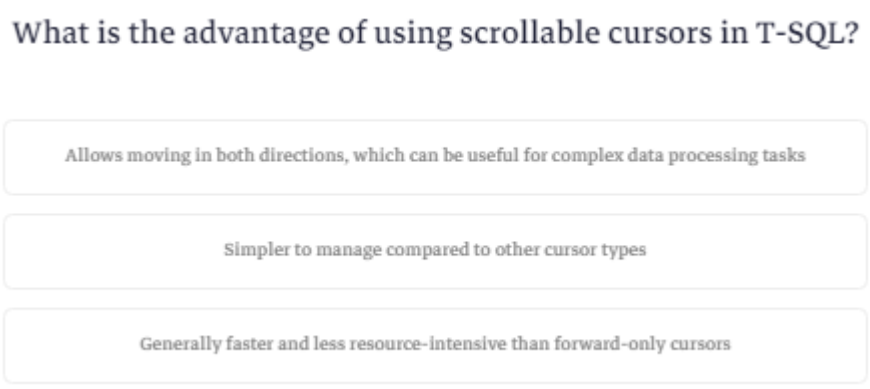
**Advantages and Disadvantages**

- Advantages:
  - Flexibility: Allows moving in both directions, which can be useful for complex data processing tasks.
  - Specific Access: Enables accessing specific rows without processing the entire result set.
- Disadvantages:
  - Performance Overhead: Generally slower and more resource-intensive than forward-only cursors due to their flexibility.
  - Complexity: More complex to manage compared to simpler cursor types.

**Conclusion**

Scrollable Cursors in T-SQL are powerful tools for complex data retrieval and navigation scenarios. They offer great flexibility but at the cost of increased resource usage and potential performance overhead. It's important to weigh these factors and consider if a scrollable cursor is the best solution for the task at hand, or if a simpler cursor or a set-based approach could suffice. Understanding the right context and usage is key to effectively leveraging scrollable cursors in database operations.

## Quiz 5



When is it ideal to use scrollable cursors?

When you need to access data in a non-sequential order

When you need to quickly retrieve large amounts of data

When you only need to access data in a sequential order

What is a disadvantage of using scrollable cursors?

They are generally faster and less resource-intensive than forward-only cursors

They are simpler to manage compared to other cursor types

They have a potential performance overhead

What is an advantage of using simpler cursor types?

Increased resource usage

Flexibility in data processing tasks

Faster performance compared to scrollable cursors

What should you consider when deciding to use scrollable cursors?

Complex data retrieval and navigation scenarios

Resource usage is not important

Any cursor type can be used for any task

## Common Table Expressions (CTEs) in T-SQL

ال cte اختصار ل common table expressions وهيا تعتبر زي ال subqueries ومن خلالها تقدر تقسم ال queries عشان تكون مفهومه اكتر

وهيا بتختلف عن ال subquery في انها اسهل في القراءة وبتختلف عن ال temporary table في ان الداتا اللي جايه منها بتتخزن في الmemory انما ال temporary table ده بيتخزن في الداتابيز

يفضل انك تستخدمها الا لو كانت ال dataset كبيره هنا هتكون ابطأ

تقدر تستخدمها في ال recursion بس ليها حدود وممكن تعملك stack overflow

تعالي نعمل الجدول ده الأول عشان نجرب عليه

```
CREATE TABLE Employees6 (
    EmployeeId INT PRIMARY KEY,
    Name VARCHAR(100),
    Sales DECIMAL(10, 2),
    Department VARCHAR(50)
);

INSERT INTO Employees6 (EmployeeId, Name, Sales, Department) VALUES
(1, 'John Doe', 50000.00, 'Sales'),
(2, 'Jane Smith', 40000.00, 'Sales'),
(3, 'Alice Johnson', 60000.00, 'Marketing'),
(4, 'Bob Williams', 45000.00, 'Sales'),
(5, 'Mike Brown', 55000.00, 'IT');
```

تعالي كده نعمل subquery

```
select * from(
select EmployeeID,Name,Sales
from Employees6 where
Employees6.Department='Sales'
)SalesStaff
```

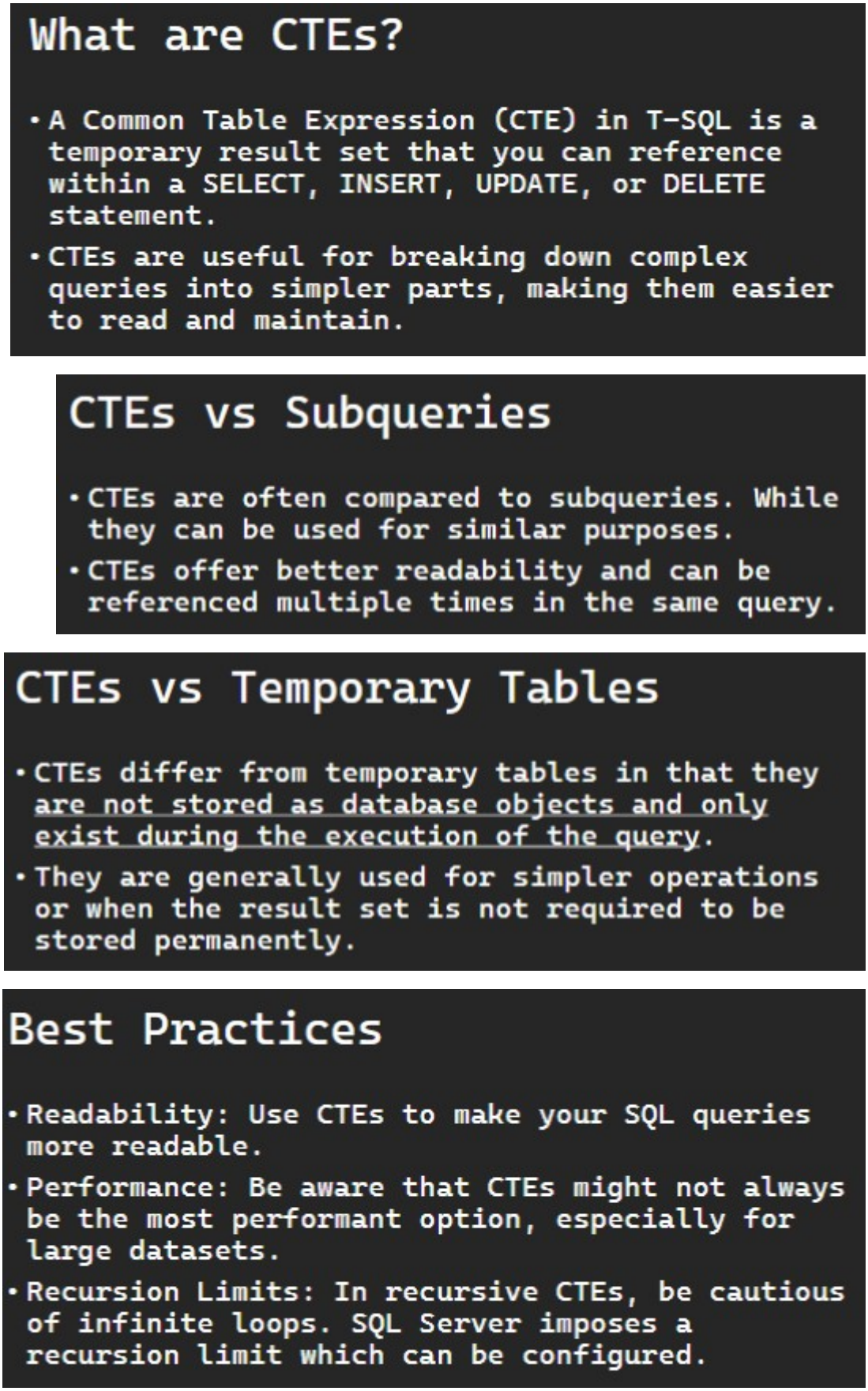
تعالي بقي نحط ال subquery اللي فاتت في cte

هنعملها ازاي؟

اصعب شيء في الدنيا

هتكتب with وبعدها اسم ال cte وبعدين as وتفتح قوسين وتحط فيهم ال subquery بتاعتك

```
with SalesStaff as(
select EmployeeID,Name,Sales
from Employees6 where
Employees6.Department='Sales'
)
select * from SalesStaff
```




# Common Table Expressions (CTEs) in T-SQL

**Introduction to CTEs**

A Common Table Expression (CTE) in T-SQL is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs are useful for breaking down complex queries into simpler parts, making them easier to read and maintain.

### Basic Syntax of a CTE

**The basic syntax of a CTE is as follows:**

```
WITH CTE_Name (Column1, Column2, ...) AS
(
    -- CTE query definition
)
-- Query using the CTE
```

### Creating a Simple CTE

A simple CTE defines a temporary result set that you can use in a subsequent query. For example:

```
WITH SalesStaff AS
(
    SELECT EmployeeId, Name, Sales
    FROM Employees6
    WHERE Department = 'Sales'
)
SELECT * FROM SalesStaff;
```

In this example, `SalesStaff` is the CTE that selects employees from the 'Sales' department. The main query then selects all records from this CTE.

### CTEs vs Subqueries

CTEs are often compared to subqueries. While they can be used for similar purposes, CTEs offer better readability and can be referenced multiple times in the same query.

### CTEs vs Temporary Tables

CTEs differ from temporary tables in that they are not stored as database objects and only exist during the execution of the query. They are generally used for simpler operations or when the result set is not required to be stored permanently.

**Best Practices**

- Readability: Use CTEs to make your SQL queries more readable.
- Performance: Be aware that CTEs might not always be the most performant option, especially for large datasets.
- Recursion Limits: In recursive CTEs, be cautious of infinite loops. SQL Server imposes a recursion limit which can be configured.

**Conclusion**

CTEs in T-SQL provide a way to structure and simplify complex SQL queries. They are particularly useful for recursive operations and can greatly enhance the readability and maintainability of your SQL code. However, it's important to use them judiciously, keeping in mind their impact on performance and their limitations compared to other SQL constructs like subqueries and temporary tables.

## Recursive CTE Example 1

تعالي اطبعلي الأرقام من 1 ل 10 باستخدام ال recursion اللي بيوفره ال cte

هنعملها ازاي؟

فاكر ال union اللي كنا بنستخدمها عشان نجمع ال records من جدولين في جدول واحد ؟

اه

طيب مش ال cte دي بيتكتب جواها query ؟

اه

احنا بقي هنكتب query ترجعلنا الرقم 1 وندمجها مع query تانيه بتستخدم اللي راجع من ال cte في انها تضيف عليه 1 وبشرط انه اللي راجع ده يكون اقل من 10

مش فاهم

بص عالكود ده

وخلي بالك انه هنا بياخد اخر رقم في الجدول اللي خارج ويحطه مكان ال number

```
WITH Numbers AS (
   SELECT 1 AS Number
   UNION ALL
   SELECT Number + 1 FROM Numbers WHERE Number < 10
)
SELECT * FROM Numbers
```

Recursive Common Table Expressions (CTEs):

- Purpose: Create loops within the execution of a query using a self-referencing CTE.

Syntax:

```
WITH recursive_cte AS
(
    -- Anchor member (base case)
    UNION ALL
    -- Recursive member (iterative case)
)
SELECT * FROM recursive_cte;
```

Example:

SQL

```
WITH Numbers AS (
    SELECT 1 AS Number
    UNION ALL
    SELECT Number + 1 FROM Numbers WHERE Number < 10
)
SELECT * FROM Numbers;
```

**Note:**

- T-SQL doesn't have a "FOR" loop like some other programming languages.
- WHILE loops and recursive CTEs provide the necessary looping capabilities.
- Use loops judiciously in T-SQL as they can impact performance, especially with large datasets.
- Consider alternative set-based approaches (e.g., using cursors or table-valued functions) when appropriate.

## Understanding Recursive CTEs for Building Employee Hierarchies

هناخد مثال تاني علي ال recursive calls في ال cte

تعالي نعمل الجدول ده الأول

```sql
CREATE TABLE Employees7 (
    EmployeeID INT PRIMARY KEY,
    ManagerID INT NULL,
    Name VARCHAR(50)
);

INSERT INTO Employees7 (EmployeeID, ManagerID, Name)
VALUES
    (1, NULL, 'CEO'),
    (2, 1, 'VP of Sales'),
    (3, 1, 'VP of Marketing'),
    (4, 2, 'Sales Manager'),
    (5, 2, 'Sales Representative'),
    (6, 3, 'Marketing Manager'),
    (7, 4, 'Sales Associate'),
    (8, 6, 'Marketing Specialist'),
       (9, 1, 'VP IT');
```

هنا الجدول ده بيمثل الهيكل الوظيفي

فالعمود بتاع الmanager id هوا بيشاور علي العمود بتاع ال employee id في نفس الجدول يعني ال Sales Manager ال manager id بتاعه 2 اللي لو جيت بصيت عليه في عمود ال employee id هتلاقيه ال 'VP of Marketing' وceo مالوش مدير

ايوه يعني عايز ايه؟

عايز اعرض بيانات الجدول وهزود عليه عمودين

اول عمود هيكون ال hierarchy وده هتكتب فيه جميع المديرين اللي فوق الموظف لحد ماتوصل لل ceo

وتاني عمود هيكون ال level وده هيكون عدد المديرين اللي فوق الموظف

زي كده

| EmployeeID | ManagerID | Name | Hierarchy | Level |
|---|---|---|---|---|
| 1 | NULL | CEO | CEO | 0 |
| 9 | 1 | VP IT | CEO -> VP IT | 1 |
| 3 | 1 | VP of Marketing | CEO -> VP of Marketing | 1 |
| 6 | 3 | Marketing Manager | CEO -> VP of Marketing -> Marketing Manager | 2 |
| 8 | 6 | Marketing Specialist | CEO -> VP of Marketing -> Marketing Manager -> M... | 3 |
| 2 | 1 | VP of Sales | CEO -> VP of Sales | 1 |
| 4 | 2 | Sales Manager | CEO -> VP of Sales -> Sales Manager | 2 |
| 7 | 4 | Sales Associate | CEO -> VP of Sales -> Sales Manager -> Sales Asso... | 3 |
| 5 | 2 | Sales Representative | CEO -> VP of Sales -> Sales Representative | 2 |

طيب هنعملها ازاي؟

تعالي نمسكها واحده واحده

اول حاجه هنجهز ال cte

```
with employeeHierarchy as(

union all
)
select * from employeeHierarchy
order by Hierarcy
```

بعد كده هنلاقي نفسنا محتاجين اتنين query واحده هنثبتها عند record معين عشان دي هتكون نقطة البدايه في كل مره هنلف فيها عالجدول

فهنختار ال record بتاع ال ceo عشان ده اللي مالوش مدير وده القمه بتاعت الhierarchy ونزود عليه عمودين

```
select EmployeeID,ManagerID ,Name ,
Hierarcy=cast (name as varchar(max)),
Level=0 from Employees7
where Employees7.ManagerID is null

union all

)
select * from employeeHierarchy
order by Hierarcy
```

طب تاني query ؟

دي هتكون ال query اللي هتستخدم اللي طالع من ال cte عشان يشغل ال recursion

هيكون عباره عن انه يربط الجدول بتاع ال employees7 بالجدول اللي ناتج عن ال cte ويحط نفس الاعمده بس هيغير في القيمه بتاعت ال hierarchy وال level

```
with employeeHierarchy as(
select EmployeeID,ManagerID ,Name ,
Hierarcy=cast (name as varchar(max)),
Level=0 from Employees7
where Employees7.ManagerID is null

union all

select e.EmployeeID,e.ManagerID ,e.Name ,
```

```
Hierarcy= eth.Hierarcy+'->'+e.Name,
Level=eth.Level+1
from
Employees7 e inner join employeeHierarchy eth
on e.ManagerID=eth.EmployeeID
)
select * from employeeHierarchy
order by Hierarcy
```

طيب دي كده اشتغلت ازاي تاني برضه معلش؟

انت دلوقتي عندك 2 queries واحده بتجيب ال record ده

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |

والتانيه هتلف معنا لفه حلوه

ازاي؟

هيا ال query التانيه بتقول ايه؟

ال query التانيه اللي هيا دي

```
select e.EmployeeID,e.ManagerID ,e.Name ,
Hierarcy= eth.Hierarcy+'->'+e.Name,
Level=eth.Level+1
from
Employees7 e inner join employeeHierarchy eth
on e.ManagerID=eth.EmployeeID
```

بتقولي هاتلي ال records اللي في جدول ال employees7 وزود عليهم عمودين بشرط ان ال manager id اللي في جدول ال employees7 يكون مساوي لل employee id الناتجه عن ال cte

يعني كأن الداتا اللي خارجه من ال cte دي خاصه بالمديرين والداتا اللي خارجه من ال employees دي خاصه بالموظفين فبنقوله هاتلنا الموظف اللي مديره رقمه كذا

فتعالي نمسك ال records بتاعت جدول ال employees7

ده الجدول اهول

| | EmployeeID | ManagerID | Name |
|---|---|---|---|
| 1 | 1 | NULL | CEO |
| 2 | 2 | 1 | VP of Sales |
| 3 | 3 | 1 | VP of Marketing |
| 4 | 4 | 2 | Sales Manager |
| 5 | 5 | 2 | Sales Representative |
| 6 | 6 | 3 | Marketing Manager |
| 7 | 7 | 4 | Sales Associate |
| 8 | 8 | 6 | Marketing Specialist |
| 9 | 9 | 1 | VP IT |

اول record اللي هوا ال ceo ال manager id بتاعه ب null

طب لحد دلوقتي كل الداتا اللي راجعاه هيا الrecord بتاع اول query اللي هوا ال ceo برضه

هل ال employee id بتاع ال ceo بيساوي null ؟

لا

يبقي نتيجة اول record ب null وبالتالي مش هنحطها والجدول النهائي هيكون فيه ال record اللي نتج عن اول query بس

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |

تعالي نشوف تاني record اللي هوا ال VP of Sales

ال manager id بتاعه يساوي 1

هل ال query اللي خارجه من ال cte فيها record ال employee id فيها يساوي 1 ؟

اه عندي ال ceo ال employee id بتاعه ب 1

قالك خلاص يبقي هاتلي ال record ده من جدول ال employees وزوده علي الجدول اللي خارج من ال cte

وتعالي في عمود ال level هتحط فيه ال level اللي خارج من الcte وزود عليه واحد (اللي خارج من ال cte كان صفر هنزود عليه واحد هيبقي قيمتة ال level بواحد)

وتعالي علي عمود ال hierarchy وخد ال ceo اللي خارجه من ال cte وزود عليها vp of sales

يطلعلك نتيجة ال cte كده

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |
| 2 | 2 | 1 | VP of Sales | CEO->VP of Sales | 1 |

هنيجي بعدها لثالث record وهوا ال vp of marketing ال manager id بتاعه بيساوي 1

هل عندك في الجدول اللي خارج من ال cte موظف ال id بتاعه ب 1 ؟

اه عندي ال ceo

يبقي هاتلي ال record بتاع ال vp of marketing وزوده عالجدول اللي خارج من ال cte

وهتيجي عال level وتزوده بواحد

طب هنا هناخد ال level بتاع ال ceo ولا بتاع ال vp of sales ؟

انت عامل الشرط بتاع ال inner join ايه؟

عامله انه رقم المدير في ال employees7 يساوي رقم الموظف في الcte

طيب انهي موظف في ال cte رقمه 1 ؟

ال ceo

يبقي انت هتزود واحد علي رقم ال level بتاعت ال ceo

وتقوم جاي في ال hierarchy وتحط ال ceo وبعدها ال vp of marketing زي ماعملت المره اللي فات

يصبح الجدول اللي خارج من ال cte بالمنظر ده

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |
| 2 | 2 | 1 | VP of Sales | CEO->VP of Sales | 1 |
| 3 | 3 | 1 | VP of Marketing | CEO->VP of Marketing | 1 |

بعدها نيجي لل sales manager

ال manager id بتاعه يساوي 2

هل فيهrecord في الجدول اللي خارج من ال cte ال employee id بتاعه يساوي 2 ؟

اه عندي ال vp of sales

خلاص يبقي هاتلي ال record بتاع ال sales manager وزوده عالجدول اللي خارج من ال cte

وتعالي عال level وزوده بواحد

طب انا هنا هاخد انهي level بالظبط؟

قولتلك قبل كده انت حاطط شرط ال join ايه؟

انه رقم الموظف في الجدول اللي خارج من ال cte يساوي رقم المدير بتاع ال record اللي في جدول ال employees7

طب انهي مدير ال employee id بتاعه يساوي 2 ؟

ال vp of sales

خلاص يبقي هتاخد ال level بتاع ال vp of sales اللي هوا بيساوي 1 وتزود عليه واحد يبقي 2

وهتيجي عال hierarchy بتاعت ال vp of sales اللي هيا CEO->VP of Sales وتزود عليها sales manager

فيصبح الجدول اللي خارج من ال cte بالمنظر ده

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |
| 2 | 2 | 1 | VP of Sales | CEO->VP of Sales | 1 |
| 3 | 3 | 1 | VP of Marketing | CEO->VP of Marketing | 1 |
| 4 | 4 | 2 | Sales Manager | CEO->VP of Sales->Sales Manager | 2 |

وتفضل مكمل نفس الخطوات لحد ماتوصل للنتيجه دي

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |
| 2 | 2 | 1 | VP of Sales | CEO->VP of Sales | 1 |
| 3 | 3 | 1 | VP of Marketing | CEO->VP of Marketing | 1 |
| 4 | 4 | 2 | Sales Manager | CEO->VP of Sales->Sales Manager | 2 |
| 5 | 5 | 2 | Sales Representative | CEO->VP of Sales->Sales Representative | 2 |
| 6 | 6 | 3 | Marketing Manager | CEO->VP of Marketing->Marketing Manager | 2 |
| 7 | 7 | 4 | Sales Associate | CEO->VP of Sales->Sales Manager->Sales Associate | 3 |
| 8 | 8 | 6 | Marketing Specialist | CEO->VP of Marketing->Marketing Manager->Marketi... | 3 |
| 9 | 9 | 1 | VP IT | CEO->VP IT | 1 |

وبعدين رتبه من حيث ال hierarchy هيطلع معاك كده

| | EmployeeID | ManagerID | Name | Hierarcy | Level |
|---|---|---|---|---|---|
| 1 | 1 | NULL | CEO | CEO | 0 |
| 2 | 9 | 1 | VP IT | CEO->VP IT | 1 |
| 3 | 3 | 1 | VP of Marketing | CEO->VP of Marketing | 1 |
| 4 | 6 | 3 | Marketing Manager | CEO->VP of Marketing->Marketing Manager | 2 |
| 5 | 8 | 6 | Marketing Specialist | CEO->VP of Marketing->Marketing Manager->Marketi... | 3 |
| 6 | 2 | 1 | VP of Sales | CEO->VP of Sales | 1 |
| 7 | 4 | 2 | Sales Manager | CEO->VP of Sales->Sales Manager | 2 |
| 8 | 7 | 4 | Sales Associate | CEO->VP of Sales->Sales Manager->Sales Associate | 3 |
| 9 | 5 | 2 | Sales Representative | CEO->VP of Sales->Sales Representative | 2 |

This lesson explains how to use a recursive Common Table Expression (CTE) in T-SQL to build and display a hierarchical employee structure.

**What are CTEs?**

CTEs are temporary result sets defined within a T-SQL query. They act like temporary tables or views and can be used to break down complex queries into smaller, more manageable steps. Recursive CTEs can iterate upon themselves, allowing them to handle complex hierarchical structures like employee organizations.

**Example:**

**We have the following data:**

| | EmployeeID | ManagerID | Name |
|---|---|---|---|
| 1 | 1 | NULL | CEO |
| 2 | 2 | 1 | VP of Sales |
| 3 | 3 | 1 | VP of Marketing |
| 4 | 4 | 2 | Sales Manager |
| 5 | 5 | 2 | Sales Representative |
| 6 | 6 | 3 | Marketing Manager |
| 7 | 7 | 4 | Sales Associate |
| 8 | 8 | 6 | Marketing Specialist |
| 9 | 9 | 1 | VP IT |

**We need to write query using CTE to retrieve it as tree and the output will be like this:**

| EmployeeID | ManagerID | Name | Hierarchy | Level |
|---|---|---|---|---|
| 1 | NULL | CEO | CEO | 0 |
| 9 | 1 | VP IT | CEO -> VP IT | 1 |
| 3 | 1 | VP of Marketing | CEO -> VP of Marketing | 1 |
| 6 | 3 | Marketing Manager | CEO -> VP of Marketing -> Marketing Manager | 2 |
| 8 | 6 | Marketing Specialist | CEO -> VP of Marketing -> Marketing Manager -> Marketing Specialist | 3 |
| 2 | 1 | VP of Sales | CEO -> VP of Sales | 1 |
| 4 | 2 | Sales Manager | CEO -> VP of Sales -> Sales Manager | 2 |
| 7 | 4 | Sales Associate | CEO -> VP of Sales -> Sales Manager -> Sales Associate | 3 |
| 5 | 2 | Sales Representat... | CEO -> VP of Sales -> Sales Representative | 2 |

**The Query is:**

```sql
WITH EmployeeTreeHierarchy AS (
    -- Anchor member: This selects the root of the hierarchy (CEO in this case) and starts at Level 0
    SELECT EmployeeID,
    ManagerID, Name,
    CAST(Name AS VARCHAR(MAX)) AS 'Hierarchy', 0 AS Level
    FROM Employees
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive member: This part of the CTE builds the hierarchy and increments the Level by 1
    SELECT e.EmployeeID, e.ManagerID, e.Name,
    CAST(ETH.Hierarchy + ' -> ' + e.Name AS VARCHAR(MAX)),
    ETH.Level + 1 AS Level
    FROM Employees e
    INNER JOIN EmployeeTreeHierarchy ETH ON e.ManagerID = ETH.EmployeeID
)
-- This SELECT statement retrieves the hierarchical data with Level
SELECT * FROM EmployeeTreeHierarchy
ORDER BY Hierarchy;
```

## Generating a Date Series Using CTE

ده مثال تاني علي ال recursive calls وهوا اني مثلا لو عايز اطبع كل التواريخ من اول 1/1/2023 لحد 31/1/2023

اظن بعد الشرح اللي كتبته فوق أصبحت الأمور اسهل

تعالي الأول نعرف 2 variables واحد بيمثل تاريخ البدايه والتاني بيمثل تاريخ النهاية

```sql
declare @StartDate date ='2023/1/1';
declare @EndDate date ='2023/1/31';
```

بعدين هنجهز ال cte ونقطة البدايه اللي هيا ال anchor

```sql
with DateSeries AS(
select @StartDate as DateValue
union all

)
select * from DateSeries
```

كده فاضل ال query بتاعت ال recursion

عاوزينه في كل مره ياخد اخر تاريخ من cte ويزود عليه يوم لحد مايوصل لل end date

```sql
select DATEADD(DAY,1,DateValue)
from DateSeries
where DateValue<@EndDate
```

وده الكود كله

```sql
declare @StartDate date ='2023/1/1';
declare @EndDate date ='2023/1/31';

with DateSeries AS(
select @StartDate as DateValue
union all
select DATEADD(DAY,1,DateValue)
from DateSeries
where DateValue<@EndDate
)
select * from DateSeries;
```

## Generating a Date Series Using CTE in T-SQL

Objective:

Learn to use a Common Table Expression to generate a continuous series of dates, which is a common requirement in reporting and data analysis, especially when dealing with time series data.

Scenario:

Suppose we need to generate a report that includes every day within a specific date range, even if some dates don't have corresponding data in our database.

SQL Query with CTE:

```sql
DECLARE @StartDate DATE = '2023-01-01'; -- Start of the date range
DECLARE @EndDate DATE = '2023-01-31';   -- End of the date range



WITH DateSeries AS (
    -- Anchor member: Start with the initial date
    SELECT @StartDate AS DateValue



    UNION ALL



    -- Recursive member: Add one day in each iteration
    SELECT DATEADD(day, 1, DateValue)
    FROM DateSeries
    WHERE DateValue < @EndDate
)
SELECT DateValue
```

```
FROM DateSeries;
```

# Identifying Duplicate Records Using CTE in T-SQL

تعالي نعمل الجدول ده الأول

```
CREATE TABLE Contacts (
    ContactID INT IDENTITY(1,1) PRIMARY KEY,
    Name NVARCHAR(100),
    Email NVARCHAR(100)
);

INSERT INTO Contacts (Name, Email) VALUES
('John Doe', 'john.doe@example.com'),
('Jane Smith', 'jane.smith@example.com'),
('Alice Johnson', 'alice.johnson@example.com'),
('Bob Ray', 'bob.ray@example.com'),
('Charlie Brown', 'charlie.brown@example.com'),
('David Smith', 'david.smith@example.com'),
('Eva Green', 'eva.green@example.com'),
('Frank White', 'frank.white@example.com'),
('Gina Hall', 'gina.hall@example.com'),
('Hank Marvin', 'hank.marvin@example.com'),
('Irene Adler', 'irene.adler@example.com'),
('Jane Smith', 'jane.smith@example.com'), -- Duplicate
('Karl Marx', 'karl.marx@example.com'),
('Lena Horne', 'lena.horne@example.com'),
('Mike Tyson', 'mike.tyson@example.com'),
('Nina Simone', 'nina.simone@example.com'),
('Oscar Wilde', 'oscar.wilde@example.com'),
('Zelda Kim', 'david.smith@example.com'), ---- Duplicate
('Peter Pan', 'peter.pan@example.com'),
('Quincy Jones', 'quincy.jones@example.com'),
('Rachel Green', 'rachel.green@example.com'),
('Steve Jobs', 'steve.jobs@example.com'),
('Tim Cook', 'tim.cook@example.com'),
('Uma Thurman', 'uma.thurman@example.com'),
('Vince Neil', 'vince.neil@example.com'),
('Walter White', 'walter.white@example.com'),
('Xena Warrior', 'xena.warrior@example.com'),
('Yoko Ono', 'yoko.ono@example.com'),
('Zelda Fitzgerald', 'zelda.fitzgerald@example.com'),
('David Smith', 'david.smith@example.com'); -- Duplicate
```

عاوزين بقي نعمل ايه؟

عايزين نطلع كل ال records اللي فيها ايميلات متكرره

لو جينا نعملها بال query العاديه هنعملها كده

```
select C.ContactID,C.Name,C.Email from Contacts C
inner join
( select Contacts.Email,Duplicated=count(*) from Contacts
group by Email
having count(*)>1
)as t
ON C.Email=t.Email
```

شايف انت بقي ال sub query اللي بعد inner join ؟

هناخدها نحطها في cte

بس كده

```
with DuplicatedEmails as(

select Contacts.Email,Duplicated=count(*)
from Contacts
group by Email
having count(*)>1
)
select C.ContactID,C.Name,C.Email from Contacts C
inner join DuplicatedEmails
ON C.Email=DuplicatedEmails.Email
```

## Ranking Items Using CTE

تعالي نعمل الجدول ده

```
CREATE TABLE SalesRecords (
    RecordID INT IDENTITY(1,1) PRIMARY KEY,
    EmployeeID INT,
    SaleAmount DECIMAL(10, 2),
    SaleDate DATE
);

INSERT INTO SalesRecords (EmployeeID, SaleAmount, SaleDate) VALUES
(1, 1200.50, '2023-01-10'),
(2, 800.00, '2023-01-11'),
(1, 600.25, '2023-01-15'),
(3, 950.75, '2023-01-20'),
(2, 400.00, '2023-01-21'),
(1, 300.00, '2023-01-25'),
(3, 1200.00, '2023-01-30'),
(2, 750.00, '2023-02-05'),
(1, 500.00, '2023-02-10'),
(3, 850.00, '2023-02-15'),
(4, 3000.75, '2023-02-15');
```

الجدول ده فيه مبيعات الموظفين خلال أيام الشهر

احنا بقي عايزين نحسب اجمالي مبيعات كل موظف خلال الشهر ونعملهم rank بحيث نشوف مين الاعلي ومين الأقل باستخدام ال cte

اول حاجه عاوزين نعملها اننا نعمل جدول نحسب فيه اجمالي المبيعات بتاعت كل موظف

```
with SalesTotals as(
select EmployeeID,Sum(SaleAmount)as TotalSales
from SalesRecords
group by EmployeeID
)
```

بعد كده هنعمل cte تاني يعمل ranking

```
,
RankingSales as(
select EmployeeID ,TotalSales ,RANK()over (order by
TotalSales desc)as SalesRank

from SalesTotals
```

واخر حاجه نختار ال records اللي طالعه من ال cte الأخير

وده الكود كله

```
with SalesTotals as(
select EmployeeID,Sum(SaleAmount)as TotalSales
from SalesRecords
group by EmployeeID
),
RankingSales as(
select EmployeeID ,TotalSales ,RANK()over (order by TotalSales desc)as SalesRank

from SalesTotals

)
select EmployeeID,TotalSales,SalesRank
from RankingSales
```

## Ranking Items Using CTE in T-SQL

Objective:

Learn to use a Common Table Expression to rank items in a dataset. This example will demonstrate how to assign ranks to sales employees based on their total sales.

Scenario and Dataset:

Imagine we have a table named `SalesRecords` with columns `EmployeeID`, `SaleAmount`, and `SaleDate`. We want to rank the employees based on their total sales.

SQL Query with CTE:

```
WITH SalesTotals AS (
    SELECT
        EmployeeID,
        SUM(SaleAmount) AS TotalSales
    FROM SalesRecords
    GROUP BY EmployeeID
), RankedSales AS (
    SELECT
```

```
        EmployeeID,
        TotalSales,
        RANK() OVER (ORDER BY TotalSales DESC) AS SalesRank
    FROM SalesTotals
)
SELECT EmployeeID, TotalSales, SalesRank
FROM RankedSales;
```

Components of the Query:

1. First CTE - SalesTotals:

```
SELECT
    EmployeeID,
    SUM(SaleAmount) AS TotalSales
FROM SalesRecords
GROUP BY EmployeeID
```

- 
  - Calculates the total sales for each employee.
  - Groups the sales records by `EmployeeID`.

1. Second CTE - RankedSales:

```
SELECT
    EmployeeID,
    TotalSales,
    RANK() OVER (ORDER BY TotalSales DESC) AS SalesRank
FROM SalesTotals
```

- 
  - Uses the `RANK()` window function to assign a rank to each employee based on their total sales.
  - Orders the employees in descending order of their total sales.

1. Final SELECT Statement:
    - 
    - Retrieves the `EmployeeID`, `TotalSales`, and `SalesRank` from the `RankedSales` CTE.

Key Concepts:

- Window Functions: The `RANK()` function is a type of window function used for ranking.
- Data Aggregation and Ranking: Combining grouping and ranking to derive meaningful insights from sales data.

Practical Application:

- Ranking sales employees in a company to identify top performers.
- Product ranking based on sales or customer reviews.
- Any scenario requiring a rank order based on specific criteria in a dataset.

Conclusion:

Using CTEs along with window functions in T-SQL provides a powerful tool for data analysis tasks like ranking. It allows for clear, organized, and efficient SQL queries that can derive valuable insights from complex datasets.

## Calculating Average Sales of Top Performing Employees Using CTE

من الجدول اللي فات عايزين نختار اعلي 3 موظفين عاملين مبيعات ونحسب المتوسط بتاعهم

```
with SalesTotals as(
select EmployeeID,Sum(SaleAmount)as TotalSales
from SalesRecords
group by EmployeeID
),
topSales as(
select top 3 EmployeeID ,TotalSales
from SalesTotals
order by TotalSales desc
)
select AVG(TotalSales) from topSales
```

# Calculating Average Sales of Top Performing Employees Using CTE in T-SQL

Objective:

Learn how to use a Common Table Expression to filter and perform calculations on a subset of data. In this case, we will identify the top N sales employees based on their total sales and then calculate their average sales.

Scenario and Dataset:

Suppose we have a table `SalesRecords` with columns `EmployeeID`, `SaleAmount`, and `SaleDate`. We want to find the average sales amount of the top 3 employees based on their total sales.

SQL Query with CTE:

```
WITH TotalSales AS (
    SELECT
        EmployeeID,
        SUM(SaleAmount) AS TotalSales
    FROM SalesRecords
    GROUP BY EmployeeID
), TopSalesEmployees AS (
    SELECT TOP 3 EmployeeID, TotalSales
    FROM TotalSales
    ORDER BY TotalSales DESC
)
SELECT AVG(TotalSales) AS AverageTopSales
FROM TopSalesEmployees;
```

Components of the Query:

1. First CTE - TotalSales:

```
SELECT
    EmployeeID,
    SUM(SaleAmount) AS TotalSales
FROM SalesRecords
GROUP BY EmployeeID
```

- 
    - Calculates the total sales for each employee.
    - Groups the sales records by `EmployeeID`.

1. Second CTE - TopSalesEmployees:

```
SELECT TOP 3 EmployeeID, TotalSales
FROM TotalSales
ORDER BY TotalSales DESC
```

- 
    - Selects the top 3 employees with the highest total sales.
    - Orders employees by `TotalSales` in descending order.

1. Final SELECT Statement:

- 
    - Calculates the average sales (`AverageTopSales`) of these top-performing employees.

Key Concepts:

- Nested CTEs: Using multiple CTEs in sequence to filter and process data.
- Aggregation and Filtering: Combining aggregation (`SUM`) and top N selection (`TOP 3`) to derive insights.
- Average Calculation: Applying an average calculation (`AVG`) to a specific subset of data.

Practical Application:

- Identifying and analyzing top performers in sales teams.
- Used in scenarios where insights are needed for a specific subset of data (e.g., top customers, best-selling products).

Conclusion:

Using CTEs in T-SQL allows for clear, sequential data processing steps, making complex queries more readable and maintainable. This approach is particularly useful in scenarios requiring multiple layers of data filtering and aggregation.

## Quiz

### What does CTE stand for?

- Common Table Expression
- Complex Table Execution
- Conditional Table Evaluation
- Common Table Execution

### What can CTEs be referenced in?

- SELECT statements
- INSERT statements
- UPDATE statements
- All of the above

### How are CTEs different from subqueries?

- CTEs offer better readability
- CTEs can be referenced multiple times in the same query
- CTEs are stored as database objects
- CTEs are executed more quickly

How are CTEs different from temporary tables?

- CTEs are stored as database objects
- CTEs exist only during the execution of the query
- CTEs can be used for simpler operations
- CTEs are required to store the result set permanently

What is a best practice when using CTEs?

- Use CTEs to improve performance
- Avoid using CTEs for large datasets
- Be cautious of infinite loops in recursive CTEs
- Use CTEs instead of subqueries

**Thank You.**

لازم تطبق كويس

وممكن بعد كده نحتاج نضيف حاجات للكورس


The End