**What is T-SQL (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 databases interaction but also introduce several additional features tailored to Microsoft’s database platform.

* Transact – 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.
* It 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.
* 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.
* 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.
* It also 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.
* It also 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.
* It provides a range of security features, including the ability to define user roles, grant permissions and implement encryption. This ensures that the 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** (Transact – SQL) and **PL/SQL** (Procedural Language / SQL) are both extensions of SQL, the standard language for interacting with relational databases. While they share some similarity, 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:** Developedby Oracle Corporation for use with **Oracle Databases.**

**Variables in T – SQL**

**What is Variable?**

* Variable 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 @VraiableName DataType.
* Example: DECLARE @EmployeeName VARCHAR(50);
* Here, @EmployeeName is a variable of the type VARCHAR(a string datatype) with maximum length of 50 character.

**Assigning Values to Variable:**

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 you T – SQL code wherever you might use literals or expressions.

Example In Query:

SELCT \* 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, NUMIRIC)
  + Character types (CHAR, VARCHAR)
  + Date and Time types (DATE, DATTIME)

**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 resources consumption.
* Use descriptive names for readability.

**Conclusion:**

* Variables in T – SQL are essential for writing dynamic and flexible queries.
* The 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.

**If statement**

The IF statement in T – SQL is a control – of – flow language that 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 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 the end of a block of statement instead of the curl brackets {} in usual programming languages.

**Simple IF statement:**

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**

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**

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 Assignment**

Using IF statement with variables:

Variables can be used within 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, @Salary DECIMAL(10,2) =50000;

IF @Age > 18 AND @Salary >=50000

BEGIN

PRINT 'Eligible for loan'

END

ELSE

BEGIN

PRINT 'Not eligible for loan'

END

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

DECLARE @Grade CHAR(1) = 'B';

DECLARE @AttendancePercentage INT = 70;

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-engagment email'

END

ELSE

BEGIN

PRINT 'Customer is active'

END

**Error Handling with IF**

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

DECLARE @ErrorValue INT;

--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

**Key Characteristics of @@ERROR:**

1. Value: After each T – 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 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… CATHC: In modern T – SQL programming, @@ERROR is often replaced by the more robust TRY… CATCH constructs, 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 Transactions: In transactions 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 the TRY… CATCH block for more comprehensive error handling. Understanding @@ERROR is still useful, particularly for maintaining and understanding legacy SQL Server code.

**IF EXISTS**

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

And here how to search for a table in you database using if exists

IF EXISTS (SELECT R =1 FROM sys.tables WHERE name = 'Employees' AND schema\_id = SCHEMA\_ID('dbo'))

BEGIN

PRINT 'Yes, Employees table exists'

END

ELSE

BEGIN

PRINT 'No, Employees table does not exist'

END

You can also perform update or insertion or deletion statements

DECLARE @Name NVARCHAR (20) = 'Abdalla Samir'

DECLARE @Name2 NVARCHAR (20) = 'Abdalla Khalifa'

IF EXISTS (SELECT R=1 FROM Employees WHERE Name = @Name2)

BEGIN

DELETE FROM Employees

WHERE Name = @Name2

END

ELSE

BEGIN

IF EXISTS (SELECT R =1 FROM Employees WHERE Name = @Name)

BEGIN

UPDATE Employees

SET Name = @Name2

WHERE Name = @Name

END

ELSE

BEGIN

INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate)

VALUES (6, @Name, 4, GETDATE ())

END

END

In the above example: we have declared two variables in the first condition we check if the latest ELSE is executed and delete the record

So the statement is safe where if all condition is cannot be executed it deletes the record completely and insert it again using the insert statement written in the nested condition.

**Best Practices and Summary for IF statement**

* Keep the logic within IF statements simple better readability.
* avoid deeply nested if statements when possible. Consider alternative structures like CASE statements or stored procedures.
* Ensure condition are clear and cover al 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 check to complex decision – making processes. Understanding and utilizing IF statements effectively can greatly enhance the functionality and efficiency of you SQL scripts.

**CASE Statement**

**Introduction:**

T – SQL does not have a dedicated switch statement as 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 SWICTH 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 quires. 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:

CASE input\_expression

    WHEN expression1 THEN result1

    WHEN expression2 THEN result2

    ...

    ELSE default\_result

END

* Input\_expression: The expression to compare 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 statement 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 flexible tool for conditional logic, adaptable for various scenarios in SELECT, UPDATE and ORDER BY clauses, enhancing SQL query functionality and dynamism.

**Simple CASE**

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

SELECT

    EmployeeID,

    CASE DepartmentID

        WHEN 1 THEN 'Engineering'

        WHEN 2 THEN 'Human Resources'

        WHEN 3 THEN 'Sales'

        ELSE 'Other'

    END AS DepartmentName

FROM Employees;

**Searched CASE**

This uses searched CASE for more complex conditions, categorizing sales by sales amount.

SELECT \*,

CASE

WHEN SaleAmount <= 100 THEN 'Week'

WHEN SaleAmount BETWEEN 101 AND 200 THEN 'Good'

WHEN SaleAmount BETWEEN 201 AND 300 THEN 'Very Good'

WHEN SaleAmount > 300 THEN 'Excellent'

else 'Not Specified'

END AS SaleLevel

FROM Sales

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

This clause doesn’t sort by sales amount it checks if the amount is bigger than 150 it takes priority 1 otherwise it takes 2 and then it sorts the sales amount desc.

SELECT \* FROM Sales

ORDER BY

CASE

WHEN SaleAmount > 150 THEN 1

ELSE 2

END, SaleAmount desc;

**CASE in UPDATE**

UPDATE Employees2

SET Salary =

CASE

WHEN PerformanceRating > 90 THEN Salary \* 1.5

WHEN PerformanceRating BETWEEN 75 AND 90 THEN Salary \* 1.10

WHEN PerformanceRating BETWEEN 50 AND 74 THEN Salary \* 1.05

ELSE Salary

END;

**Nested CASE Statement**

Nested CASE statement in T – SQL allow for complex conditional logic in SQL queries.

SELECT \*,

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 the above 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 performance rating.
* Each department has different criteria for calculating the bonus.
* The bonus is a percentage of the salary, and the percentage varies based on the performance rating and the department.

**CASE Within a GROUP**

To demonstrate the use of 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 is and example SQL query to achieve this:

SELECT PerformanceCategory,

COUNT(\*) AS NumberOfEmps,

AVG(Salary) As AverageSalaries

FROM

(SELECT \* ,

CASE

WHEN PerformanceRating >=80 THEN 'High'

WHEN PerformanceRating >=60 THEN 'Medium'

ELSE 'Low'

END AS PerformanceCategory

FROM Employees2) AS PerformanceTable

Group by PerformanceCategory;

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

**While Loop**

A WHILE loop in T – SQL is a control – of – flow language constructs that allow the execution of a specified block of SQL statement 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 WHILE Loop:**

WHILE loops are often used for repetitive tasks where 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 there are no FOR or DO WHILE statements as you would fin in many other programming languages. The primary looping constructs available in T – SQL are the WHILE loop and the COURSOR, 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 statement as long as a specified condition is true.

**Bread and Continue**

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
   1. Loop is like in other programming languages, are used to execute a set of statements repeatedly until a specified condition is met.
   2. The most common loop in T – SQL us the WHILE loop.
2. BREAK Statement
   1. Purpose: it is used to immediately exit the loop, regardless of whether the loop condition is still true.
   2. Usage: Typically used when a certain condition is met inside the loop, and there is no need to continue looping.
3. CONTINUE Statement
   1. Purpose: It is used to skip the rest of the loop body and immediately start the next iteration.
   2. Usage: Commonly used to skip certain iteration based on a condition.
4. Kye Differences and Usage Scenarios:
   1. Use BREAK when you need to exit the loop entirely.
   2. Use CONTINUE when you want to skip the current iteration proceed with the next iteration.

**BEGIN and END Blocks**

They are a control – of – flow language construct used to group a series of T – SQL statement into 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.

**What they are used for:**

1. To group statements: Group multiple T – SQL statements so the 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:**

* Scoop Definition: They define the start and end of a statement group.
* Nested Blocks: These blocks can be nested inside one another.

**Error Handling TRY… CATCH**

**Introduction:**

It 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 it important:

* Prevents data corruption: Proper error handling can prevent partial updated 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**:

Is the primary mechanism for error handling.

* TRY Block: You place the code that might cause an error inside a TRY block. If an error occurs, execution is passed to the associated CATCH block.
* CTCH Block: it contains the code that run if an error occurs in the TRY block. It can log the error, rollback transaction and take other appropriate actions.

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.
* 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: Return 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 a potential error.

-- 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 EmployeeID. The error is caught in the CATCH block where a message is printed, and you could also add logic to rollback a transaction if necessary.

**Best Practices:**

* User TRY...CATCH fir 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 bout the database structure or system.

**Conclusion:**

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

**Error Functions**

SQL Server provides several functions that can be used within a CATCH block 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 to identifying the specific error that occurred.
* **ERROR\_SEVERITY:**
  + Purpose: Returns the error 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 for the error
  + Usage: Useful for providing additional information about the error 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 message of the error message.
  + Usage: Offers a detailed description of the error, which is valuable for understanding what wen wrong.

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

**THROW Statement**

The THROW statement 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 statement with example.

1. Purpose: Used to raise an exception and transfer control to a CATCH block.
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 string less than 2048 characters.
* State: A constant number between 0 and 255.
* **Key Points:**
  + If you don’t specify these arguments, the original error is passed on.
  + You cannot use THROW to throw an error that is caught be a CATCH block outside the current scoop: this means that if an error is raised using THROW within a TRY block, it must be caught by the corresponding CATCH that is in the same scoop as the TRY block. Is there’s nested TRY…CATCH an error thrown inside the inner TRY block cannot be caught be the CATCH block of the outer one. It has to be caught within the same level of nesting (you can throw multiple times if necessary).

**Example: Updating Product Inventory:**

We have 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 valid scenario in must inventory system.

  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

* In this code we first enter the 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 1.
* If the stock quantity is valid, the procedure updates the Products table with the new stock quantity.
* If an error thrown, control passes to the CATCH block, which captures and returns the error information.

This example demonstrates how the THROW can be effectively used to ensure data integrity and prevent invalid operations in database procedures. It’s constraints that are not directly enforced by the database schema itself.

**@@ERROR**

It is a system function that returns the error number of the last executed statement. It’s an older method of error checking in SQL Server, often used before the introduction of the TRY…CATCH constructs.

**Understanding @@ERROR:**

* Purpose: It provides the error number of the last executed statement. If the last statement was successful, it returns 0.
* Usage:
  + It must be checked immediately after the statement that might cause an error, because any subsequence statement will reset the method to 0 if executes successfully.
  + It is often used in older scripts (legacy code) or in older systems where TRY…CATCH is not available or applicable.

--ATTEMP TO INSERT NA INVALID RECORD INTO A TABLE

INSERT INTO Departments (DepartmentID,Name)

VALUES(1, 'BUSINESS');--ASSUME DEPARTMENT ID IS A PRIMARY KEY AND 1 ALREDY EXISTS

DECLARE @ERRORNUMBER INT = @@ERROR

IF @ERRORNUMBER <> 0

BEGIN

--HANDLE THE ERROR

PRINT 'AN ERROR OCCURRED DURING THE INSERT STATEMENT'

-- YOU CAN ALSO CAPTURE THE SPECIFIED ERROR NUMBER AND STORE IT OR USE IT IN LOGIC

PRINT 'THE ERROR NUMBER IS: '+ CAST(@ERRORNUMBER AS VARCHAR)

END

**@@ROWCOUNT**

It is a system function 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 INSERT, UPDATE, DELETE or SELECT statements.

**Understanding @@ROWCOUNT:**

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

**Syntax**

SELECT @@ROWCOUNT

**Example Usage:**

Consider 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;

* Immediately after @@ROWCOUNT is used to return the number of rows the were updated by the UPDATE statement.

**Practical Consideration:**

* 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 statement, to take different actions depending on the number of rows affected by a previous operation.
* Zero Rows Affected: If no rows 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 statement 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.

**What are Transactions**

It is a series of database operations that are treated as a single logical unit. It ensures that either all operations with it are executed or none are.

ACID Properties: Transactions 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 set of properties that guarantee that database transactions are processed reliably.

1. Atomicity: This ensure 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. Consistence: It ensures that a transaction brings the database form one valid state to another. Integrity constraint is maintained so that the database remains consistent before and after the transaction.
3. Isolation: It ensures that transactions are securely and independent processed at the same time without interference, but the results of the transaction are as if the transaction were processed sequentially. This prevents transactions from reading intermediate (and possibly inconsistent) data.
4. Durability: It guarantees that once a transaction has been committed, it will remain so, even if 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 performance 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 date 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.

SET @DESACCOUNT =2;

SET @SOURCEACCOUNT =1;

BEGIN TRANSACTION

BEGIN TRY

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID =@SOURCEACCOUNT;

IF @@ROWCOUNT = 0

THROW 50001,'ACCOUNT UPDATE THE SOURCE ACCOUNT', 1

UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID =@DESACCOUNT;

IF @@ROWCOUNT = 0

THROW 50001,'COULDNOT UPDATE THE DESTINATION ACCOUNT', 1

INSERT INTO Transactions (FromAccount, ToAccount, Amount, Date) VALUES (@SOURCEACCOUNT,@DESACCOUNT,100,GETDATE());

IF @@ROWCOUNT = 0

THROW 50001,'COULD NOT LOG THE TRANSACTION', 1

COMMIT;

END TRY

BEGIN CATCH

ROLLBACK

SELECT ERROR\_MESSAGE() AS ERRORMESSAGE,

ERROR\_NUMBER() AS ERRORNUMBER,

ERROR\_STATE() AS ERRORSTATE

END CATCH

This is example demonstrates transferring 100$ from account to another and log the operation onto the Transactions table.

* The BEGIN TRANSACTION starts the transaction.
* BEGIN TRY…BEGIN CATCH handles any errors, rolling back the transaction if necessary.
* COMMIT confirms the transaction.
* ROLLBACK undoes it in case of errors.

**Table Variables**

They 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 functions 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 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 table variable is limited to the batch, stored procedure, of function in which it is defined. This can simplify transaction management and error handling.

**Differences Between Table Variable and Temporary Table:**

* 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, 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 datasets.

**Best Practices:**

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

**Conclusion:**

Table variables provide a convenient way to temporary store and manipulate small sets of data. They are particularly useful for quick operations and 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.

--DECLARE VARIABLE TABLE NAME @EmployeesTable

--THIS TABLE VARIABLE IS STORED IN MEMORY

--AND IS SCOPED TO THE PATCH, STORED PROCEDURE OR FUNCTION

DECLARE @EmployeesTable TABLE

(

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(50)

);

-- INSERT RECORD INTO THE TABLE

INSERT INTO @EmployeesTable (EmployeeID, Name, Department) VALUES (10,'Mohammed', 'Marketing');

INSERT INTO @EmployeesTable (EmployeeID, Name, Department) VALUES (11,'Ali', 'Sales');

-- QUERY THE TABLE

SELECT \* FROM @EmployeesTable

UPDATE @EmployeesTable SET Name ='ABDLALLA' WHERE EmployeeID =10;

--NO NEED TO DROP THE TABLE

--IT WILL AUTOMATICALLY GOES OUT OF SCOPE AND DEALLOCATED AT THE END OF THE EXECUTION PATH

**Temporary Tables**

They 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 #TemptTable (…).
3. Global Temporary Tables: Created with a double hash (#) symbol.  
   Visible to all connections and are deleted when the last connection using 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 creates 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:**

--CREATE A LOCAL TEMPORARY TABLE NAME #EMPLOYEESTEMP

--THIS TABLE WILL BE STORED IN THE tempdb DATABASE AND IS VISIBLE ONLY TO THIS SESSION

CREATE TABLE #EMPLOYEESTEMP

(

EmployeeID INT,

NAME NVARCHAR(100),

DEPARTMENT NVARCHAR(100)

);

INSERT INTO #EMPLOYEESTEMP (EmployeeID, NAME, DEPARTMENT) VALUES (10,'MOHAMMED', 'MARKETING')

INSERT INTO #EMPLOYEESTEMP (EmployeeID, NAME, DEPARTMENT) VALUES (11,'ALI', 'SALES')

SELECT \* FROM #EMPLOYEESTEMP WHERE DEPARTMENT= 'SALES'

UPDATE #EMPLOYEESTEMP SET NAME = 'ABDALLA' WHERE DEPARTMENT = 'SALES'

SELECT \* FROM #EMPLOYEESTEMP WHERE DEPARTMENT= 'SALES'

DROP TABLE #EMPLOYEESTEMP

--DROP THE TEMPORARY TABLE THIS IS A GOOD PRACTICE

-- ALTHOUGH THE TABLE WOULD AUTOMATICALLY BE DELETED WHEN THE SESSION ENDS

--THIS IS A GLOBAL TABLE

CREATE TABLE ##EMPLOYEESTEMP

(

EmployeeID INT,

NAME NVARCHAR(100),

DEPARTMENT NVARCHAR(100));

* **Differences between Temp Table vs Variable Table**

In T – SQL, which is the extension for SQL, two common ways to store data temporarily are through temporary tables and table variables.

Here’s a lesson that outlines the diffs between them:

* Definition and Scope:
  + Temporary Tables: Created using the CREATE TABLE statement, with 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.
  + Tables Variables: Declared using the DECLARE statement and have a similar structure to permanent tables. The syntax is DECLARE @TableName (column definition). They have a limited scope and are typically used within the function, stored procedure or batch in which they are declared.
* Lifetime:
  + Temporary Tables: Exists until they are explicitly dropped using the DROP TABLE command or until the session/connection that created them is closed.
  + Table Variable: 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 storin joins with other tables. They support indexes, statistics and can result in better query performance for large datasets.
  + Table Variable: 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.
* Used 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 Variable: Useful for quick, temporary storage of a small amount of data that does not require transactional rollbacks or heavy – duty operations.

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

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

The difference between temporary and normal (permanent) tables are significant in terms of scoop, life span, usage and physical storage.

1. **Life Span and Scope:**
   * Temporary Tables: They are created in the tempdb database and exists only for the duration of this session or connection that created them. Local temporary tables are visible only to the connection that created them, while global temporary table 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 to any user. They are visible an accessible to any user with 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 dataset manipulations.
   * Normal Tables: Permanent tables are stored in the database they are created and are subject to more extensive logging and locking. This ensure 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 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 Table: They have the 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 can’t 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 tables and normal tables depends on the specific requirements of the task at hand. Temporary tables are ideal for transient data and quick, session – specified operations, whereas normal tables are suited for storing persistent data that requires full transactional support, backup and recovery.

**Stored Procedures**

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 and additional layer of security by recreating direct access to the data.
* Maintainability: centralizing business logic in stored procedures makes changes easier and more consistent.

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

They 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 procedure:

* SQL Queries and DML Statements: This includes SELEC, INSERT, UPDATE, DLETE and MERGE statements for data querying and manipulation.
* Variables Declarations and Assignments: You can declare local variables using the DECLARE statement and set values with the SET or SELECT statement.
* Control Flow Statement:
  + 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 (through 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 Procedure 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 int T – SQL.
* System Stored Procedures and Function 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.
* RAISEERROR 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 Statement: Such as CREATE, ALTER or DROP typically for temporary objects or within dynamic SQL.
* XML Handling: XML data manipulation 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 the best practices while writing stored procedures, 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 specific version’s documentation for the most accurate information.

**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 stored procedure named SP\_AddNewPerson in SQL Server. This stored procedure is designed to add a new person’s record to a table name 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 with 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 maximum length of 100 characters.
  + @LastName NVARCHAR (100): This parameter accepts the last name, also as an NVARCHAR string with maximum length of 100 characters.
  + @Email NVARCHAR (255): This parameter accepts the email address, allowing a string with length 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 statement 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 are taken form 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 () 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 person to the People table and return the identity of the new row through the 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;

Or go to your database -> Programmability -> Stored Procedures -> System Stored Procedures -> mouse write click -> then choose Execute Stored Procedure: it will generate command to execute the stored procedure.

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.

**READ: Get All People**

CREATE PROCEDURE SP\_GetAllPeople

AS

BEGIN

    SELECT \* FROM People

END

To execute the SP\_GetAllPeople, which been defined to select and return all records from the People table, we can use a simple EXEC (execute) command in T – SQL. This command tells SQL Server to run the named stored procedure.

EXEC SP\_GetAllPeople;

This command will execute the SP\_GetAllPeople procedure, which in turn runs the SQL query written in it. As a result, it will return all rows from the People table, including all columns.

**READ: Get Person By ID 1st way:**

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 2nd way:**

The other way is to create SP\_GetPersonByID 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 SP\_GetPersonByID procedure, you can include additional output parameter that indicates whether a record was found. This parameter can be Boolean or an integer flag.

**The stored procedure:**

CREATE PROCEDURE SP\_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 PersonID = @PersonID)

    BEGIN

        SELECT

            @FirstName = FirstName,

            @LastName = LastName,

            @Email = Email

        FROM People

        WHERE PersonID = @PersonID;

        SET @IsFound = 1;

    END

    ELSE

    BEGIN

        SET @IsFound = 0;

    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 table.
* If the record is found, it retrieves the details and sets the flag to 1.
* If not, the flag will be 0.

**Here’s how to execute this stored procedure:**

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 if the value of @Found if 1 then the person was found, and you can retrieve their details else this means the person was not found.

**UPDATE: Update a Person's Details**

CREATE PROCEDURE SP\_UpdatePerson

    @PersonID INT,

    @FirstName NVARCHAR(100),

    @LastName NVARCHAR(100),

    @Email NVARCHAR(255),

@RowsAffected INT OUTPUT

AS

BEGIN

    UPDATE People

    SET FirstName = @FirstName, LastName = @LastName, Email = @Email

    WHERE PersonID = @PersonID

SET @RowsAffected = @@ROWCOUNT

END

**Execute the SP:**

DECLARE @RowsAffected INT;

EXEC SP\_UpdatePerson

    @PersonID = 1,

    @FirstName = 'UpdatedFirstName',

    @LastName = 'UpdatedLastName',

    @Email = 'updated.email@example.com',

@RowsAffected = @RowsAffected OUTPUT

SELECT @RowsAffected AS RowsAffected;

**DELETE:**

CREATE PROCEDURE SP\_DeletePerson

@PersonID INT,

@ROWSAFFECTED INT OUTPUT

AS

BEGIN

DELETE FROM People

WHERE PersonID = @PersonID;

SET @ROWSAFFECTED = @@ROWCOUNT

END

**Execute:**

DECLARE @rowsaffected int;

EXEC Sp\_DeletePerson

@PersonID =6 ,

@rowsaffected = @ROWSAFFECTED OUTPUT

Select @rowsaffected as ROWSAFFECTED

select \* from People

**RETURN Statement:**

**Introduction:**

The stored procedures can return a value to the calling environment. This is an essential feature for communicating the success or failure for 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 the 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 or non – zero values for various error conditions or specific status).
* Also return is used to exit from SP.

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

Stored Procedure to check if a person exists in the People table, to check for the existence of a person based on a unique identifier, such as PersonID.

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 table.
* The IF EXISTS statement checks for the existence of 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 the stored procedure, you would call it with a specific PersonID and check the return value.

DECLARE @Result INT;

EXEC @Result = SP\_CheckPersonExists @PersonID = 123;

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:**

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

**Conclusion:**

Return values in 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 Stored Procedure:**

DROP PROCEDURE ‘SP\_NAME’;

**Sp\_helptext:**

The sp\_helptext command 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 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 example how to use it:**

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:

Retrieving the Text Definition of a stored procedure.

EXEC sp\_helptext 'YourStoredProcedureName'

Replace 'YourStoredProcedureName' with the name of the stored procedure, function, trigger, view, or user – defined function that you want to examine. This command will display the text definition of the stored procedure in the query results.

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 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 SQL objects. It helps developers and administrators gain insights into the logic and implementation of these database objects.

**String Functions**

1. LEN
   * Purpose: Returns the number of characters in a specific string.
2. UPPER
   * Purpose: Converts all characters in a specific string to uppercase.
3. LOWER
   * Purpose: Converts all characters in a specific string to lowercase.
4. SUPSTRING
   * Purpose: Returns part of a string, starting at a specified position and for specified length.
5. CHARINDEX
   * Purpose: Returns the starting position of the first occurrence of a specified expression in a string.
6. REPLACE
   * Purpose: Replace all occurrences of a specified string value within a given string with another string value.
7. LTRIM
   * Purpose: Removes leading spaces from the string
8. RTRIM
   * Purpose: Removes trailing spaces from a string.
9. TRIM
   * Purpose: Removes both leading and trailing spaces from a string.
10. CONCAT
    * Purpose: Concatenates two or more strings into one string.
11. LEFT
    * Purpose: Returns the left part of a string with the specified number of characters.
12. RIGHT
    * Purpose: Returns the right part of a string with the specified number of characters.
13. DIFFERENCE:
    * Purpose: Returns an integer from 0 to 4 indicating how strong the similarity between two strings.

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.

**This is the official Microsoft documentation:**

[**https://learn.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-ver16**](https://learn.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-ver16)

this comprehensive resource provides detailed description, 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, REPLECATE, 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).

**Date Functions**

It’s crucial for manipulating and querying datetime data. These functions allow you to format dates, calculate time intervals, and extract specific parts of a date.

Commonly used functions:

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, moths, years, etc.).
4. DATEPART ()
   * Purpose: Returns a specified part of a date, such as a year, month, days, etc.
5. DATENAME ()
   * Purpose: Returns the name of the specified part of the date, such as name of the month, day of the week, etc.
6. DAY ()
   * Purpose: Extract the day part of a date.
7. MONTH ()
   * Purpose: Extracts the month 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 las day of the month for a specified date.
12. SYSDATETIME ()
    * Purpose: Returns the system date and time with functional 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.

Microsoft documentation:

<https://learn.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql?view=sql-server-ver16>

**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)

**Window Functions**

Also known as windowed or analytical functions. It allows 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 function:**

* 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\_NUBER (), 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 ().

**ROW\_NUMBER ():**

It assigns 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., tow 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 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 order 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.

**RANK () Function:**

They are used to provide sequential numbering of the rows in the result set. The RANK () function is one 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

This is 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 comes 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;

* We use the RANK () function within the 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

This function is useful for ranking rows in a dataset. In our case, it helps in understanding the relative performance of students based on their grade.

**Difference Between RANK and DENSE\_RANK:**

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 order 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:

* + Assign 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, assign 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 if incremented by one. For example, if two students are tied for rank 1, the next student will receive rank 2.

**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 gabs in ranking after ties. Use DENSE\_RANK when you want a continuous ranking sequence without gaps.

**RANK with PARTITION BY**

The PARTITION BY clause is used in conjunction with window functions. 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.

**Concepts PARTITION BY clause:**

This clause divides the result set into partitions where the ranking function restarts its count for each partition.

Example:

SELECT \*

, RANK() OVER (PARTITION BY Subject ORDER BY Grade DESC) AS GradeRank FROM Students;

* 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 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 will increment based on the total number of students with the previous grade within that subject.
* The rank resets for each subject.

Conclusion

Using RANK with PARTITION BY allows for more nuance analysis of data, enabling ranking within specific subsets of data. This is particularly useful in scenarios where comparative ranking is required within categories.

**Aggregate Functions with Partition**

Window functions provide a way to perform calculations across set 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.

**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 calculated 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 order 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.

**Exploring LAG and LEAD Functions**

LAG and LEAD functions are invaluable for comparing data across rows without complex joins. These window functions fetch data form 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.

Concepts:

* LAG: Retrieves data form a preceding row int the result set.
  + Used for comparing the current row with past data.
* LEAD: Retrieves data form 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;

* 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 data set ordered by Grade DESC.

Understanding 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 order by Grade DESC, so the comparison is in the context of descending grades.

Conclusion:

Combining LAG and LEAD in a single query allows for comprehensive comparison within the dataset, revealing patterns and relationships that might not be immediately apparent.

**Paging using OFFSET and FETCH NEXT:**

Paging is a critical feature in database management, particularly useful when dealing with large data sets. It allows for displaying data in segmented chunks, enhancing performance and user experience. SQL Server offer 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.

Concepts

* 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) \* @RowsPersPage 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 provides a straightforward method to implement paging. Which is essential for managing and displaying large volumes of data efficiently.

**Scalar Functions**

They provide a way to encapsulate reusable logic and returns 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.

They can be used in T – SQL queries in any part of the query where an expression is allowed.

**Explanation:**

* We have two tables: Students and Teachers, which store information about students and teachers, respectively.

**Explanation:**

* We have two tables: Students and Teachers, which store information about students and teachers, respectively.

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 Functions name GetAverageGrade.
* The function takes a parameter @subject of type NVARCHAR(50) to specify the subject for which we want to calculate the average grade for.
* Inside the function, we declare a variable @averageGrade to hold the calculated average grade.
* We use the SELECT statement to calculate the average grade form the students table, filtered by the specified subject.
* Finally, we return the @averageGrade as the result of the function.

**Using the Scalar Function:**

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 we understand how to create and use Scalar Functions, we 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 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 teacher whose subject’s average grade is grater than 80.

**Conclusion:**

It’s 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 our T – SQL code, we can modularize and reuse custom logic, making our queries more efficient and maintainable.

**Table Valued Functions**

They are user – defined functions that returns tabular data as a result. They are a powerful feature 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 them

**Types of Table – values – Functions:**

Theres 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.

1. **Inline Table – Valued – Functions (ITVFs):**

It is user – defined function that returns a table expression. They are defined using the RETUNRNS TABLE clause and a single SELECT statement ITVFs provides a flexible way to encapsulate complex logic and reuse it in queries.

You can’t use UPDATE or INSERT statements in it they are designed to be read – only, they can’t modify the data in the database, which means you can’t perform data manipulation operations like INSERT, UPDATE, DELETE, or MERGE within them.

The primary purpose of an ITVFs 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 critical aspect for functions in SQL Server. If you nee to perform insert or update operations, you would typically use stored procedures or other types that are designed for such purpose.

**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 ITVF name GetStudentBySubject that accepts a parameter called @Subject of type NVARCHAR.
* 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 ITVF:**

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 the rows from the table – valued function GetStudentBySubject(‘Math’). This will give us all the students studying math.
* In Example 2, we calculate the average of students studying Science by using the AVG function and passing the TVF as the input.

**Conclusion:**

ITVFs 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.

**Using ITVFs with JOIN:**

Now, let's see how we can use Inline Table-Valued Functions in JOIN operations to combine data from multiple tables.

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 this example, we used the SELECT statement to retrieve data from the students TVF GetStudentBySubject and join it with the teachers table base on the shared subject.
* We specify alias 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, TeacherName and Grade.

Using the ITVFs with JOIN operations allows us to combine data from multiple tables efficiently.

**Multi-Statement Table-Valued Functions (MTVFs):**

They offer a powerful way to encapsulate complex logic and return tabular data. MTVFs can be used to perform multiple statements, manipulate data, and return the result set as a table.

**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 example, we create a MTVFs name GetTopPerformingStudents.
* The function returns a table with the column StudentID, Name, Subject, 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 3 performing students from the student table, ordered by Grade in descending order.
* Finally, we return the 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 MTVFs 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 Exploration:**

Now that you understand how to create and use Multi-Statement Table-Valued Functions, you can further explore the possibilities:

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 the example, we join the MTVFs 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:**

MTVFs are a powerful tool for encapsulating complex logic and returning tabular data.

**Dynamic SQL**

Stored Procedures can be used to generate and execute dynamic SQL. Dynamic SQL refers to SQL statement that are constructed and executed at runtime as string. This approach allows for a high degree of flexibility, but it also requires careful handling to avoid issued 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 executed 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

* This method supports parametrized 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: it can be vulnerable to SQL injection attacks, especially if user input is concatenated directly in SQL strings. Always validate and sanitize user inputs.
* Debugging and Maintenance: It can be harder to debug and maintain compared to static SQL, as the actual SQL executed is constructed at runtime.
* Performance: while it offers flexibility, it may not always be as performant as static SQL, especially if it leads to frequent recompilations of the SQL statement.
* Security: Ensure that the stored procedure and it 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, it’s important to use this features judiciously, considering the implications for security, performance and maintainability.

**SQL Injection**

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:**

It is a security vulnerability the occurs when the attacker manipulates user input to executer unintended SQL statements. This can lead to unauthorized access, data breaches, or even complete compromise of the database.

**Understanding the SQL injection:**

it occurs when an attacker inject malicious SQL code into an application’s input fields, bypassing input validation and directly manipulating SQL statements. Let’s consider the provided script 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 @nputb value is not properly validated or sanitized, an attacker can manipulate it to alter the original query’s logic and retrieve unintended date.

**SQL Injection Risks:**

* 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: the can execute arbitrary SQL statements, potentially gaining full control of the database.

**Preventive Measures:**

**Parametrized Queries:**

* + Use parametrized queries or prepared statements instead of concatenating user inputs.
  + Parametrized queries separate the SQL code from the user input, preventing malicious input from altering the queries 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:

* Int 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 student ids are used.
* Otherwise, an alternative action can be taken, such as displaying ab error message.

**Conclusion:**

This is a severe security vulnerability that can lead to unauthorized access, data breaches, or even complete compromise of the database.