Name: Hinda Nguyen

Date: August 27th, 2025

Course: IT FDN 130

https://github.com/hindanguyenle/DBFoundations-Module07

Assignment 07 - Functions

Contents

Introduction to UDFs

User-Defined Functions (UDFs) in SQL are an essential tool for organizing and reusing logic within queries. They allow developers to encapsulate common calculations, formatting rules, or filtering conditions into a single defined function that can be applied across multiple queries and reports. By doing so, UDFs reduce redundancy, improve readability, and promote consistency. This assignment provided hands-on practice with scalar functions, inline table-valued functions (iTVFs), and multi-statement table-valued functions (mTVFs), showing how each type contributes to efficient and maintainable database solutions.

When to Use a UDF

A SQL User-Defined Function (UDF) is valuable for encapsulating reusable logic that can be applied across multiple queries or reports. For example, in this assignment, the function **fProductInventoriesWithPreviousMonthCountsWithKPIs** was created to filter inventory records based on whether the inventory count increased, decreased, or remained the same compared to the previous month. By passing a parameter (1, 0, or -1), the function returns only the rows matching the desired KPI, eliminating the need to repeat complex logic in multiple queries. UDFs also support calculations, formatting, and aggregations. For instance, formatting product prices as US dollars using the FORMAT function in

SELECT ProductName, FORMAT(UnitPrice, 'C', 'en-US') AS PriceUSD

FROM vProducts

provided a consistent and readable output across queries.

```
CREATE OR ALTER FUNCTION fProductInventoriesWithPreviousMonthCountsWithKPIs
(
    @KPIFilter INT = NULL
)
RETURNS TABLE
AS
RETURN
(
    SELECT
        ProductName,
        InventoryDate,
        InventoryCount,
        PreviousMonthCount,
        CountVsPreviousCountKPI
    FROM vProductInventoriesWithPreviousMonthCountsWithKPIs
    WHERE @KPIFilter IS NULL OR CountVsPreviousCountKPI = @KPIFilter
);

go
```

Figure 1. ***Example of a user-defined function from this assignment.***

Functions

SQL functions extend the power of queries by allowing developers to encapsulate logic that can be applied consistently across different contexts. Functions can perform calculations, format values, or return sets of rows, making queries both easier to read and easier to maintain. In this assignment, we worked with three main types of functions: Scalar, Inline Table-Valued (iTVFs), and Multi-Statement Table-Valued (mTVFs). Each serves a distinct purpose: scalar functions return individual values, inline functions provide parameterized table-like results, and multi-statement functions support more complex logic through multiple operations. Together, they form a flexible toolkit for managing data transformations and reporting needs in SQL.

Scalar Functions

Scalar functions return a single value for each row, typically used for calculations or formatting. In this assignment, the use of FORMAT(UnitPrice, 'C', 'en-US') to display product prices in US dollars illustrates this concept. Each unit price was transformed into a readable format without altering the underlying data. Scalar functions are frequently used in SELECT, WHERE, and ORDER BY clauses to simplify query logic and improve clarity.

```
SELECT
    ProductName,
    FORMAT(UnitPrice, 'C', 'en-US') AS PriceUSD
FROM vProducts
ORDER BY ProductName;

go
```

Figure 2. ***Example of a scalar function in this assignment.***

While scalar functions operate at the row level, inline table-valued functions return entire sets of rows, making them more powerful for dataset-level operations.

Inline Table-Valued Functions (iTVFs)

Inline Table-Valued Functions return a table from a single SELECT statement, making them efficient and straightforward. They are conceptually similar to the views created in the assignment, such as **vProductInventories** and **vCategoryInventories**. For instance, **vProductInventories** combined product names with inventory counts and formatted inventory dates through a single SELECT statement, producing a reusable result set. Inline functions are particularly useful when a flexible, parameterized table-like output is required without the overhead of multiple statements.

```sql
CREATE VIEW vProductInventories
AS
SELECT
    p.ProductName,
    i.InventoryDate AS RawInventoryDate,  -- keep the original date for sorting
    FORMAT(i.InventoryDate, 'MMMM, yyyy') AS InventoryDateFormatted,
    i.Count AS InventoryCount
FROM vProducts AS p
INNER JOIN vInventories AS i
    ON p.ProductID = i.ProductID;

go
```

Figure 3. ***Example of an inline table-valued function from this assignment.***

When even more complex logic is required, such as multiple statements, variables, or conditional checks, multi-statement table-valued functions are the most appropriate choice.

Multi-Statement Table-Valued Functions (mTVFs)

Multi-Statement Table-Valued Functions allow multiple SQL statements, variable declarations, and complex logic to return a table. A clear example from the assignment is **fProductInventoriesWithPreviousMonthCountsWithKPIs** (see Figure 1), which referenced a KPI view and applied conditional logic based on InventoryCount versus PreviousMonthCount to return KPI values of 1, 0, or -1. This approach enabled dynamic reporting and filtering while keeping queries concise and maintainable. Replicating the same logic in separate queries would have been repetitive and prone to error, highlighting the efficiency of mTVFs in managing more complex requirements.

Summary

This assignment demonstrated the importance of User-Defined Functions in SQL by applying them in real scenarios. Scalar functions provided formatting and calculation capabilities for individual values, inline table-valued functions offered efficient reusable datasets similar to parameterized views, and multi-statement table-valued functions supported more complex reporting requirements through conditional logic. Collectively, these tools simplified queries, reduced redundancy, and improved maintainability. Understanding when and how to use each type of UDF is essential for building clear, efficient, and scalable database solutions.