Michael Yu

5/21/2021

IT FDN 130: Foundations of Databases & SQL

Assignment 07

<https://github.com/myu404/DBFoundations-Module07>

SQL Functions

# Introduction

This assignment provides in-depth coverage on SQL functions. Functions similar to views and stored procedures in a sense that they are named objects that execute SQL code. Functions provide a layer of abstraction between the client user and database tables.

For a given provider (Microsoft SQL Server, Oracle, etc.), there are built-in/pre-defined functions that are ready to be used. There is also support for user-defined functions (UDFs) that allows the user to define functions for the database.

When functions are defined, they are stored to the associated database and ready to be called. Saving to database promotes reusability of the code for many users and eliminate the need to write repetitive code in script files.

# User-Defined Functions (UDFs)

User-defined functions (UDFs) refer to functions declared and defined by the end-user. These functions are not native to the server provider.

The database schema or namespace should always be a prefix to UDF name when defining the UDF. This is to avoid compiler confusion and errors when executing SQL code.

UDFs are commonly created for reporting and processing data.

Data reporting UDFs are similar to SQL views because a function is defined to report data from a table through SELECT statements. However, the advantage UDFs have over SQL views is that UDFs can provide dynamic data reporting, whereas views are static once created. UDFs can be defined with SELECT statements written in terms of UDF parameters. These parameters accept input arguments from the user and return data based on the input arguments.

Data processing UDFs are more commonly used than data reporting UDFs. Data processing UDFs are commonly defined for ETL (extract, transform, load) data processing.

ETL is common practice when migrating data over from one database to another. It is common for a user to restructure or reformat the schema from an old database to meet the schema requirements of the new database. Because this task is performed quite often, UDFs are defined to extract the data from the old database, transform the data to match the new database schema, and load the updated data schema to the new database.

Data extraction would typically involve extracting data from a table through SELECT statements and place it in a temp data. Data transformation modifies data as required to match requirements and load the transformed data to a new table.

# SQL Function

Functions are designed to return a single value or table of values. Functions that return a single value are called “scalar functions.” Functions that return a table of values are called “table functions.”

Table functions are further categorized as inline (simple) table functions and multi-statement (complex) table functions.

## Scalar Functions

Scalar functions are designed to execute SQL code defined in its function body and returns a single value. The return value could be used in a subsequent expression. It is common to call a scalar function to return a value within a SELECT query. Scalar functions are also useful for data validation such as calling a scalar function within a CHECK constraint.

If scalar functions are used within a SQL SELECT query, then the scalar function returns a single value associated to the row of data that called the scalar function.

## Inline/Simple Table Functions

Inline table functions refer to functions defined with a single SQL SELECT query and return a table based on the query. Because an inline function has a single SQL query, a table schema or table column definition is not required after declaring “Table” keyword in the RETURNS clause. The returned table schema and columns are generated implicitly based on the SQL query.

## Multi-Statement/Complex Table Functions

Multi-statement functions refer to functions defined with multiple SELECT statements and/or branching logic. These functions do not simply return a table based on a simple SQL query.

The RETURNS clause returns a table variable that is referenced within the function itself. Additionally, a table schema or table columns must be defined for the table variable in the RETURNS clause. The schema defines the table format that will be returned to the function caller. A return statement at the end of the function body is commonly provide to indicate the end of the function and it is returning the table variable back to the function caller.

## Function Differences

In general, scalar functions return single values that could be used in SQL expressions and logic (i.e. CHECK constraint). On the other hand, table functions return a table of values that could be used for reporting data.

Table functions are further categorized into inline table functions and multi-statement table functions.

Inline table functions contain simple SQL query in its code and returns a table that implicitly define the table schema based on the SQL query.

Multi-statement table functions contain multiple SQL queries and logic that returns a table based on the code execution. Furthermore, a table schema must be defined for the multi-statement table function because the table schema is not implicitly known at compile-time due to the function containing multiple SQL statements and logic branching that may affect the table output.

# Module 07 Assignment

Module 7’s assignment provides the opportunity to create SQL views and functions.

See Appendix for SQL code.

# Summary

This assignment extensively covers SQL functions. Functions are named containers that group SQL code together. Functions can be defined to return a single value (scalar functions) or table of values (table functions). Table functions could be defined with a single SELECT statement (inline/simple table function) or with code logic and multiple SELECT statements (multi-statement/complex table function).

Generally, users can define UDFs for data reporting and processing.

Dynamic data reports are accomplished through functions by passing input arguments to a function that have parameters defined. The dynamic data reports is an advantage that functions have over views.

Functions are also utilized to process data, especially when migrating data and schema matching.

# Appendix

## Assignment07DB\_YuMichael.sql

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

-- Title: Assignment07

-- Author: Michael Yu

-- Desc: This file demonstrates how to use Functions

-- Change Log: When,Who,What

-- 2021-05-15,MYu,Created File

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

Begin Try

    Use Master;

    If Exists(Select Name From SysDatabases Where Name = 'Assignment07DB\_YuMichael')

     Begin

      Alter Database [Assignment07DB\_YuMichael] set Single\_user With Rollback Immediate;

      Drop Database Assignment07DB\_YuMichael;

     End

    Create Database Assignment07DB\_YuMichael;

End Try

Begin Catch

    Print Error\_Number();

End Catch

go

Use Assignment07DB\_YuMichael;

-- Create Tables (Module 01)--

Create Table Categories

([CategoryID] [int] IDENTITY(1,1) NOT NULL

,[CategoryName] [nvarchar](100) NOT NULL

);

go

Create Table Products

([ProductID] [int] IDENTITY(1,1) NOT NULL

,[ProductName] [nvarchar](100) NOT NULL

,[CategoryID] [int] NULL

,[UnitPrice] [money] NOT NULL

);

go

Create Table Employees -- New Table

([EmployeeID] [int] IDENTITY(1,1) NOT NULL

,[EmployeeFirstName] [nvarchar](100) NOT NULL

,[EmployeeLastName] [nvarchar](100) NOT NULL

,[ManagerID] [int] NULL

);

go

Create Table Inventories

([InventoryID] [int] IDENTITY(1,1) NOT NULL

,[InventoryDate] [Date] NOT NULL

,[EmployeeID] [int] NOT NULL

,[ProductID] [int] NOT NULL

,[ReorderLevel] int NOT NULL -- New Column

,[Count] [int] NOT NULL

);

go

-- Add Constraints (Module 02) --

Begin  -- Categories

    Alter Table Categories

     Add Constraint pkCategories

      Primary Key (CategoryId);

    Alter Table Categories

     Add Constraint ukCategories

      Unique (CategoryName);

End

go

Begin -- Products

    Alter Table Products

     Add Constraint pkProducts

      Primary Key (ProductId);

    Alter Table Products

     Add Constraint ukProducts

      Unique (ProductName);

    Alter Table Products

     Add Constraint fkProductsToCategories

      Foreign Key (CategoryId) References Categories(CategoryId);

    Alter Table Products

     Add Constraint ckProductUnitPriceZeroOrHigher

      Check (UnitPrice >= 0);

End

go

Begin -- Employees

    Alter Table Employees

     Add Constraint pkEmployees

      Primary Key (EmployeeId);

    Alter Table Employees

     Add Constraint fkEmployeesToEmployeesManager

      Foreign Key (ManagerId) References Employees(EmployeeId);

End

go

Begin -- Inventories

    Alter Table Inventories

     Add Constraint pkInventories

      Primary Key (InventoryId);

    Alter Table Inventories

     Add Constraint dfInventoryDate

      Default GetDate() For InventoryDate;

    Alter Table Inventories

     Add Constraint fkInventoriesToProducts

      Foreign Key (ProductId) References Products(ProductId);

    Alter Table Inventories

     Add Constraint ckInventoryCountZeroOrHigher

      Check ([Count] >= 0);

    Alter Table Inventories

     Add Constraint fkInventoriesToEmployees

      Foreign Key (EmployeeId) References Employees(EmployeeId);

End

go

-- Adding Data (Module 04) --

Insert Into Categories

(CategoryName)

Select CategoryName

 From Northwind.dbo.Categories

 Order By CategoryID;

go

Insert Into Products

(ProductName, CategoryID, UnitPrice)

Select ProductName,CategoryID, UnitPrice

 From Northwind.dbo.Products

  Order By ProductID;

go

Insert Into Employees

(EmployeeFirstName, EmployeeLastName, ManagerID)

Select E.FirstName, E.LastName, IsNull(E.ReportsTo, E.EmployeeID)

 From Northwind.dbo.Employees as E

  Order By E.EmployeeID;

go

Insert Into Inventories

(InventoryDate, EmployeeID, ProductID, ReorderLevel, [Count])

Select '20170101' as InventoryDate, 5 as EmployeeID, ProductID, ReorderLevel, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Union

Select '20170201' as InventoryDate, 7 as EmployeeID, ProductID, ReorderLevel, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Union

Select '20170301' as InventoryDate, 9 as EmployeeID, ProductID, ReorderLevel, ABS(CHECKSUM(NewId())) % 100 as RandomValue

From Northwind.dbo.Products

Order By 1, 2

go

-- Adding Views (Module 06) --

Create View vCategories With SchemaBinding

 AS

  Select CategoryID, CategoryName From dbo.Categories;

go

Create View vProducts With SchemaBinding

 AS

  Select ProductID, ProductName, CategoryID, UnitPrice From dbo.Products;

go

Create View vEmployees With SchemaBinding

 AS

  Select EmployeeID, EmployeeFirstName, EmployeeLastName, ManagerID From dbo.Employees;

go

Create View vInventories With SchemaBinding

 AS

  Select InventoryID, InventoryDate, EmployeeID, ProductID, ReorderLevel, [Count] From dbo.Inventories;

go

-- Show the Current data in the Categories, Products, and Inventories Tables

Select \* From vCategories;

go

Select \* From vProducts;

go

Select \* From vEmployees;

go

Select \* From vInventories;

go

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Questions and Answers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

--'NOTES------------------------------------------------------------------------------------

-- 1) You must use the BASIC views for each table.

-- 2) Remember that Inventory Counts are Randomly Generated. So, your counts may not match mine

-- 3) To make sure the Dates are sorted correctly, you can use Functions in the Order By clause!

------------------------------------------------------------------------------------------'

-- Question 1 (5% of pts): What built-in SQL Server function can you use to show a list

-- of Product names, and the price of each product, with the price formatted as US dollars?

-- Order the result by the product!

Select

    [ProductName]

    , Format([UnitPrice], 'C', 'en-US') As [Product Price]

From dbo.vProducts

Order by [ProductName];

go

-- Question 2 (10% of pts): What built-in SQL Server function can you use to show a list

-- of Category and Product names, and the price of each product,

-- with the price formatted as US dollars?

-- Order the result by the Category and Product!

Select

    [CategoryName]

    , [ProductName]

    , Format([UnitPrice], 'C', 'en-US') As [Product Price]

From dbo.vProducts as p

    Inner Join dbo.vCategories as c

    On p.CategoryID = c.CategoryID

Order by [CategoryName], [ProductName];

go

-- Question 3 (10% of pts): What built-in SQL Server function can you use to show a list

-- of Product names, each Inventory Date, and the Inventory Count,

-- with the date formatted like "January, 2017?"

-- Order the results by the Product, Date, and Count!

Select

    [ProductName]

    , DateName(month, [InventoryDate]) + ', ' + DateName(year, [InventoryDate]) As [Inventory Date]

    , [Count]

From dbo.vProducts as p

    Inner Join dbo.vInventories as i

    On p.ProductID = i.ProductID

Order by [ProductName], [InventoryDate], [Count];

go

-- Question 4 (10% of pts): How can you CREATE A VIEW called vProductInventories

-- That shows a list of Product names, each Inventory Date, and the Inventory Count,

-- with the date FORMATTED like January, 2017? Order the results by the Product, Date,

-- and Count!

Create View vProductInventories

As

    Select Top 1000000000

        [ProductName]

        , DateName(month, [InventoryDate]) + ', ' + DateName(year, [InventoryDate]) As [Inventory Date]

        , [Count]

    From dbo.vProducts as p

        Inner Join dbo.vInventories as i

        On p.ProductID = i.ProductID

    Order by [ProductName], [InventoryDate], [Count];

go

-- Check that it works: Select \* From vProductInventories;

Select \* From vProductInventories;

go

-- Question 5 (10% of pts): How can you CREATE A VIEW called vCategoryInventories

-- that shows a list of Category names, Inventory Dates,

-- and a TOTAL Inventory Count BY CATEGORY, with the date FORMATTED like January, 2017?

Create View vCategoryInventories

As

    Select

        [CategoryName]

        , DateName(month, [InventoryDate]) + ', ' + DateName(year, [InventoryDate]) As [Inventory Date]

        , Sum([Count]) As [Total Count]

    From vCategories

        Inner Join vProducts

            On vCategories.CategoryID = vProducts.CategoryID

        Inner Join vInventories

            On vProducts.ProductID = vInventories.ProductID

    Group By [CategoryName], [InventoryDate]

go

-- Check that it works: Select \* From vCategoryInventories;

Select \* From vCategoryInventories;

go

-- Question 6 (10% of pts): How can you CREATE ANOTHER VIEW called

-- vProductInventoriesWithPreviouMonthCounts to show

-- a list of Product names, Inventory Dates, Inventory Count, AND the Previous Month

-- Count? Use a functions to set any null counts or 1996 counts to zero. Order the

-- results by the Product, Date, and Count. This new view must use your

-- vProductInventories view!

Create View vProductInventoriesWithPreviousMonthCounts

As

    Select Top 1000000000

        [ProductName]

        , DateName(month, [InventoryDate]) + ', ' + DateName(year, [InventoryDate]) As [Inventory Date]

        , Sum([Count]) As [Total Count]

        , IIF(Year([InventoryDate]) = 1996, 0, IsNull(Lag(Sum([Count]), 1) Over(Order by [ProductName], Month([InventoryDate])), 0)) As [Previous Count]

    From dbo.vProducts as p

        Inner Join dbo.vInventories as i

        On p.ProductID = i.ProductID

    Group By [ProductName], [InventoryDate]

    Order by [ProductName], [InventoryDate], [Total Count];

go

-- Check that it works: Select \* From vProductInventoriesWithPreviousMonthCounts;

Select \* From vProductInventoriesWithPreviousMonthCounts;

go

-- Question 7 (20% of pts): How can you CREATE one more VIEW

-- called vProductInventoriesWithPreviousMonthCountsWithKPIs

-- to show a list of Product names, Inventory Dates, Inventory Count, the Previous Month

-- Count and a KPI that displays an increased count as 1,

-- the same count as 0, and a decreased count as -1? Order the results by the

-- Product, Date, and Count!

Create View vProductInventoriesWithPreviousMonthCountsWithKPIs

As

    Select Top 1000000000

        \*

        , Case

            When [Total Count] > [Previous Count] Then 1

            When [Total Count] = [Previous Count] Then 0

            When [Total Count] < [Previous Count] Then -1

          End

        As [Count Change KPI]

    From vProductInventoriesWithPreviousMonthCounts

    Order by [ProductName], DatePart(month, [Inventory Date]), [Total Count];

go

-- Important: This new view must use your vProductInventoriesWithPreviousMonthCounts view!

-- Check that it works: Select \* From vProductInventoriesWithPreviousMonthCountsWithKPIs;

Select \* From vProductInventoriesWithPreviousMonthCountsWithKPIs;

go

-- Question 8 (25% of pts): How can you CREATE a User Defined Function (UDF)

-- called fProductInventoriesWithPreviousMonthCountsWithKPIs

-- to show a list of Product names, Inventory Dates, Inventory Count, the Previous Month

-- Count and a KPI that displays an increased count as 1, the same count as 0, and a

-- decreased count as -1 AND the result can show only KPIs with a value of either 1, 0,

-- or -1? This new function must use you

-- ProductInventoriesWithPreviousMonthCountsWithKPIs view!

-- Include an Order By clause in the function using this code:

-- Year(Cast(v1.InventoryDate as Date))

-- and note what effect it has on the results.

Create Function dbo.fProductInventoriesWithPreviousMonthCountsWithKPIs(@kpi int)

    Returns @results Table

            (

                [ProductName] sql\_variant

                , [Inventory Date] sql\_variant

                , [Total Count] sql\_variant

                , [Previous Count] sql\_variant

                , [Count Change KPI] sql\_variant

            )

    As

        Begin

            If(@kpi >= -1 And @kpi <= 1)

                Insert Into @results

                    Select Top 10000000

                    \*

                    From dbo.vProductInventoriesWithPreviousMonthCountsWithKPIs

                    Where [Count Change KPI] = @kpi

                    Order By Year(Cast([Inventory Date] as Date))

            Else

                Insert Into @results Values (null, null, null, null, null)

        Return

        End

go

/\* Check that it works:

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(1);

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(0);

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(-1);

\*/

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(1);

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(0);

Select \* From fProductInventoriesWithPreviousMonthCountsWithKPIs(-1);

go

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/