Database Tips and Tricks

Advanced SQL Puzzles

Scott Peters

https://advancedsqlpuzzles.com/

Last Updated: 07/12/2022

Welcome, in this PDF you will find the instructions and developer comments pertaining to the scripts in the following GitHub repository:

AdvancedSQLPuzzles/Database Tips and Tricks

These scripts are helpful tips and tricks that I find myself using on a routine basis. Please feel free to contact me via the contact page on my website. I would be happy to answer any questions and any feedback is welcome.

This document contains the following instructions:

1. Creating a Calendar Table
2. Performing Data Profiling
3. Pivoting Data
4. Subtracting Two Dates
5. Full Outer Joins

Creating a Calendar Table

AdvancedSQLPuzzles/Database Tips and Tricks/Calendar Table

Here is a nifty trick for creating a Calendar table (or in this case, a Calendar view) using a table valued function. We typically use table-valued functions as parameterized views. In comparison with stored procedures, the table-valued functions are more flexible as we can use them wherever tables are used.

The concept behind a Calendar table is that each entry is a date and additional columns are provided that represent complex date calculations that you would otherwise need to perform manually in your SQL statement.

Inside this GitHub repository you will find the following SQL scripts:

FnReturnCalendarTable.sql
- A script that creates the table valued function **FnReturnCalendarTable**

FnReturnCalendarTable Example Use.sql
- A script that provides an example use of the function **FnReturnCalendarTable**

1) The example usage script will create a table, **dbo.CalendarDaysTemp** and populate with one year's worth of dates
2) It will then join to the function, **dbo.FnReturnCalendarTable** and return the calculated fields
3) It then creates the view **dbo.VwCalendarTable** to show how you can use the Calendar table in your schema

The basic usage of the **FnReturnCalendarTable** is the following:

```
SELECT  *
FROM    FnReturnCalendarTable(GETDATE());
```

The current function returns 51 different fields, ranging from different styles using the CONVERT function to government fiscal dates.

The current 51 fields are:

DateKey, CalendarDate, DateStyle1, DateStyle2, DateStyle3, DateStyle4, DateStyle5, DateStyle6, DateStyle7, DateStyle10, DateStyle11, DateStyle12, DateStyle23, DateStyle101, DateStyle102, DateStyle103, DateStyle104, DateStyle105, DateStyle106, DateStyle107, DateStyle110, DateStyle111, DateStyle112, CalendarMonth, CalendarDay, CalendarYear, CalendarWeek, CalendarQuarter, FirstDayOfQuarter, LastDayOfQuarter, MonthLongName,MonthShortName, MonthOfQuarter, FirstDayOfMonth, LastDayOfMonth, WeekOfMonth, WeekOfQuarter, DayOfWeekName, DayOfWeekNameShort, DayOfWeekNumber, IsWeekday, DayOfQuarter, DayOfMonth, GovtFiscalYear, GovtFiscalYearStartDate, GovtFiscalYearEndDate, GovtFiscalQuarter, GovtFiscalMonth, GovtFiscalDay, ISOYear, ISOWeekNumber.

Data Profiling

AdvancedSQLPuzzles/Database Tips and Tricks/Data Profiling

Here is a script that I wrote to perform a detailed data profile on a user defined table.

There are times when I need to find which columns have NULL markers, tabs, or commas, etc.... so I created one do-all script that performs all sorts of valuable profile checks, and feel free to customize it to fit your needs.

I have added comments to the script as best possible, I hope to add further instructions given the complexity of the code.   When first testing, I highly recommend testing on a small table first. This script can run long on large tables.

Inside this GitHub repository you will find the following SQL scripts:

Data Profiling.sql
- A script to perform the data profiling on a user define table

The script is rather robust so it may be easiest if you grab the code from the GitHub repository below and test it for yourself. Once inside the code, you will see where to input the table name into a variable.

Inside the code, note the following variables:

- @vSchema – The schema where the table to be profiled is located
- @vTableName – The table name to be profiled.
- @vWhereClause – Set a WHERE clause to limit the data
- @vColumnsToProfile – A table variable that you can use to limit the columns to profile

The profiling script will return two result sets. One with several columns with the results of the profile, and another result set of the table row count.

The 26 fields returned by the profile are the following:

ColumnName, DataType, IsCalculated, IsNullable, DefinedLength, Nulls, ZerosAndBlanks, DistinctValues, MinLength, MaxLength,AverageLength, ContainsCommaOrTab, ContainsNewLine, NeedsTrimmed, ContainsOtherWhitespace, ContainsNonStandardCharacters, TotalSum, MinValue, MaxValue, Average, StandardDeviation, SampleValue, PercentBit, PercentDate, PercentInteger, PercentNumeric

I would also recommend doing an internet search on "data profiling tools". There is an ever-increasing number of tools out there that have some interesting features (like creating test data) that are worth checking out.

Also, the Wikipedia article has a nice summary about it that I think everyone could benefit from.

Pivoting Data

Here is a stored procedure to automate the pivoting of data.

The pivot operator is a powerful feature in SQL, but often goes unused due to its complicated syntax. Here we simplify the syntax by encapsulating the pivot into a stored procedure. Under the hood the stored procedure uses XML and dynamic SQL to accomplish its goal. I find myself using this stored procedure when performing exploratory data analysis.

Inside this GitHub repository you will find the following SQL scripts:

SpPivotData.sql
- A script that creates the stored procedure **SpPivotData**

Pivot Data Examples.sql
- Example usages of the **SpPivotData** stored procedure

The **Pivot Data Examples.sql** gives several great ways to use the **SpPivotData** function.

To get an overall idea of its most basic usage, see below:

```
EXEC dbo.SpPivotData
  @vQuery = 'dbo.TestPivot',
  @vOnRows = 'TransactionType',
  @vOnColumns = 'TransactionDate',
  @vAggFunction = 'SUM',
  @vAggColumns = 'TotalTransactions';
```

Will return the following:

	TransactionType	2019-01-01	2019-01-02	2019-01-03	2019-01-04	2019-01-05
1	ATM	2	4	6	8	NULL
2	Signature	10	NULL	12	14	16

Subtracting Two Dates

AdvancedSQLPuzzles/Database Tips and Tricks/Subtracting Two Dates

Here I have two stored procedures that can accurately returns the difference between two dates.

Anyone who has used the DATEDIFF has probably been frustrated by its limitations. To solve this problem, I have created two functions to subtract dates. Same logic, one returns a table, the other returns a VARCHAR. Passing two dates to either of the functions return the 1) years, 2) days, 3) months, 4) minutes, 5) seconds, and 6) nano seconds between the two dates.

Inside this GitHub repository you will find the following SQL scripts:

FnDateDiffPartsChar.sql
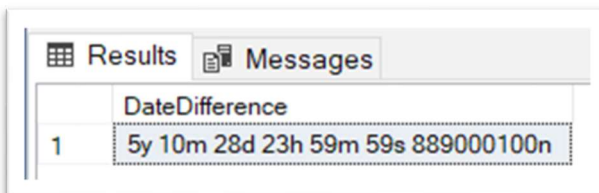- A script that creates the stored procedure **FnDateDiffPartsChar**

FnDateDiffPartsTable.sql
- A script that creates the stored procedure **FnDateDiffPartsTable**

Here are some quick examples of their usage. Note the scalar valued function is used in the SELECT statement, and the table valued function is used in the FROM statement.

```
SELECT fnDateDiffPartsChar('20170518 00:00:00.0000001','20110619 00:00:00.1110000');
```

Returns the following:

	DateDifference
1	5y 10m 28d 23h 59m 59s 889000100n

Note, you may wish to remove nanoseconds from the output. You can also modify the function to mimic the CONVERT function and pass a parameter to return a certain output style.

Running the same input on the table valued function:

```
SELECT fnDateDiffPartsTable('20170518 00:00:00.0000001','20110619 00:00:00.1110000');
```

Returns the following:

	YearDifference	MonthDifference	DayDifference	HourDifference	MinuteDifference	SecondDifference	NanoDifference
1	5	10	28	23	59	59	889000100

Full Outer Join

This script compares two identical tables using a FULL OUTER JOIN and returns a table with the results.

Recently I was helping on a project where a team was transitioning to a new customer relationship management system and trying to compare the new tables to the old. Because there were dozens of tables involved, I decided to automate the process of creating the FULL OUTER JOIN.

Inside this GitHub repository you will find the following SQL scripts:

Full Outer Join Create Sample Data (Testing).sql
- A script that creates the test data for demo purposes

Full Outer Join Insert Table Information (Part 1).sql
- A script that inserts the table information into **##TableInformation**

Full Outer Join Create Dynamic SQL (Part 2).sql
- A script that creates a dynamic SQL statement and executes

To run a quick demo, execute in order A) Testing, B) Part 1 and C) Part 2 scripts. This will create two tables **MyTable1** and **MyTable2** and do a comparison.

The script labeled "Testing" will produce the following test data:

MyTable1

TableID	CustID	Region	City	Sales	ManagerID	AwardStatus
1	453	Central	Chicago	$ 71,967.99	1324	Gold
2	532	Central	Des Moines	$ 65,232.42	6453	Gold
3	643	West	Spokane	$ 44,981.23	7542	Silver
4	643	West	Spokane	$ 44,981.23	7542	Silver
5	732	West	Helena	$ 15,232.19	8123	Bronze
6	732	West	Helena	$ 15,232.19	8123	Bronze

MyTable2

TableID	CustID	Region	City	Sales	ManagerID	AwardStatus
1	453	Central	Chicago	$ 71,967.99	1324	Gold
2	532	Central	Des Moines	$ 65,232.00	6453	Gold
3	643	West	Spokane	$ 44,981.23	7542	Bronze
4	643	West	Spokane	$ 44,981.23	7542	Bronze
5	898	East	Toledo	$ 53,432.78	9242	Silver

Looking at this data we can see several differences between the tables. Some values are different, the tables have different records, and the tables have duplicate rows. My script will accurately account for these scenarios.

Next, run the script labeled "Part 1" and populate the **##TableInformation** table with the following:

ColumnName	Value
LookupID	1
Schema1Name	dbo
Schema2Name	dbo
Table1Name	MyTable1
Table2Name	MyTable2
Exists1	CONCAT(t1.CustID, t1.Region, t1.City)
Exists2	CONCAT(t2.CustID, t2.Region, t2.City)

Because you may have multiple tables to compare, the table **##TableInformation** is used to store the information, including the schema, table names, and the join conditions.

1) The join between the two tables is created in the fields **Exists1** and **Exists2**, where a **CONCAT** function is used to group the field values together
2) Note the **t1** and **t2** difference between the fields **Exists1** and **Exists2**

Once, we have our test data setup and have inserted the table information into **##TableInformation.** Run the "Part 2" script to create the dynamic SQL statement and execute.

The dynamic SQL statement will be saved in the table **#SQLStatementFinal**. The results from this query will be saved in the temporary table ##<@vTableName1>_TemporaryTemp, which will be **##MyTable1_TemporaryTable** in our example.

To see all the field created, review the SQL statement in **#SQLStatementFinal**. For each column there is a "_Compare" field created that will be populated 1 if the fields are different, and 0 if they are the same. There is a field "Compare_Summary" in the first column of the result set that gives an overall summary if the record matches between the two tables.

The table also has information on which fields do not exist in its partner table, distinct counts, etc.… Please review the SQL statement in **#SQLStatementFinal** to get a full understanding of all fields created.

Here are some other important notes.

1) The two tables must have the exact same fields
2) Consider using views to normalize the two tables to have the exact fields
3) To change the script to use views, simply perform a search and replace on "tables" with "views" across the scripts Part 1 and Part 2
4) The script handles duplicate data by performing a distinct and then reporting the distinct count in the output

THE END