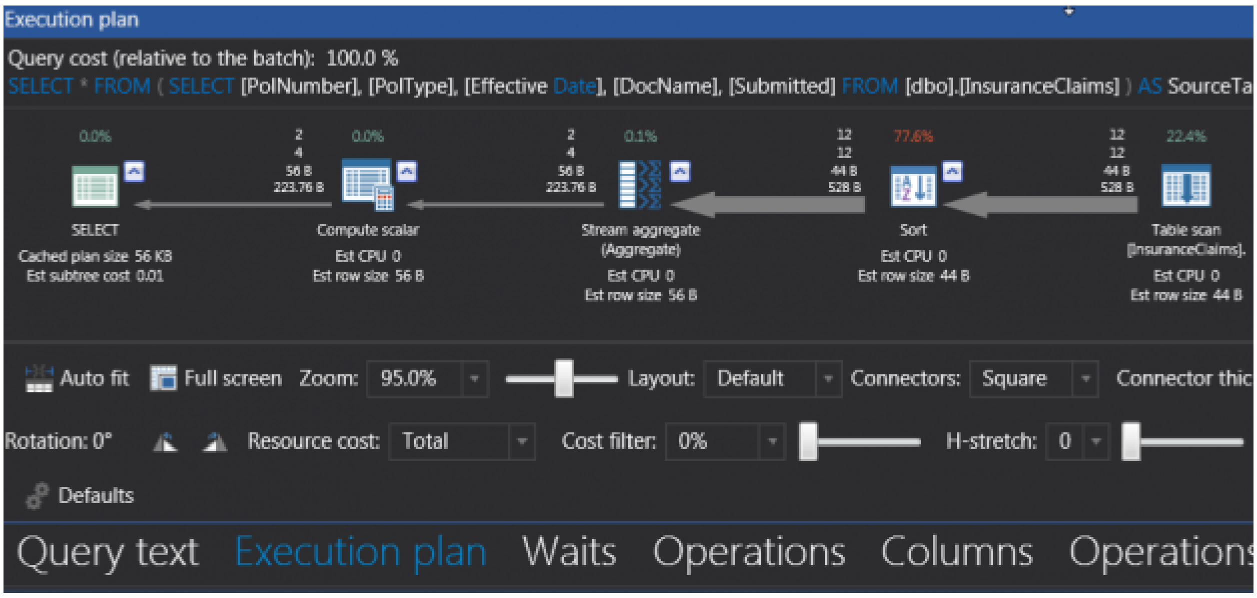
Multiple options to transposing rows into columns

sqlshack.com/multiple-options-to-transposing-rows-into-columns

Sifiso W. Ndlovu

January 4, 2016



Execution plan

Query cost (relative to the batch): 100.0 %

```
SELECT * FROM ( SELECT [PolNumber], [PolType], [Effective Date], [DocName], [Submitted] FROM [dbo].[InsuranceClaims] ) AS SourceTa
```

SELECT — 0.0% — Cached plan size 56 KB, Est subtree cost 0.01
Compute scalar — 0.0% — Est CPU 0, Est row size 56 B (2, 4, 56 B, 223.76 B)
Stream aggregate (Aggregate) — 0.1% — Est CPU 0, Est row size 56 B (2, 4, 56 B, 223.76 B)
Sort — 77.6% — Est CPU 0, Est row size 44 B (12, 12, 44 B, 528 B)
Table scan [InsuranceClaims]. — 22.4% — Est CPU 0, Est row size 44 B (12, 12, 44 B, 528 B)

Zoom: 95.0% Layout: Default Connectors: Square Rotation: 0° Resource cost: Total Cost filter: 0% H-stretch: 0

Introduction

One of the primary functions of a *Business Intelligence* team is to enable business users with an understanding of data created and stored by business systems. Understanding the data should give business users an insight into how the business is performing. A typical understanding of data within an insurance industry could relate to measuring the *number of claims received vs successfully processed claims*. Such data could be stored in source system as per the layout in *Table 1*:

Table 1: Sample policy claims data

RecKey	PolID	PolNumber	PolType	Effective Date	DocID	DocName	Submitted
1	2	Pol002	Hospital Cover	01-Oct-07	1	Doc A	0
2	2	Pol002	Hospital Cover	01-Oct-07	4	Doc B	0
3	2	Pol002	Hospital Cover	01-Oct-07	5	Doc C	1
4	2	Pol002	Hospital Cover	01-Oct-07	7	Doc D	1
5	2	Pol002	Hospital Cover	01-Oct-07	10	Doc E	1

Although each data entry in *Table 1* has a unique *RecKey* identifier, it all still relates to a single policy claim (policy *Pol002*). Thus, a correct representation of this data ought to be in a single row that contains a single instance of policy *Pol002* as shown in *Table 2*:

Table 2 Transposed layout

PolNumber	PolType	Effective Date	Doc A	Doc B	Doc C	Doc D	Doc E
Pol002	Hospital Cover	01-Oct-07	0	0	1	1	1

The objective of this article is to demonstrate different SQL Server T-SQL options that could be utilised in order to transpose repeating rows of data into a single row with repeating columns as depicted in *Table 2*. Some of the T-SQL options that will be demonstrated will use very few lines of code to successfully transpose *Table 1* into *Table 2* but may not necessary be optimal in terms query execution. Therefore, the execution plan and I/O statistics of each T-SQL option will be evaluated and analysed using ApexSQL Plan.

Option #1: PIVOT

Using a T-SQL Pivot function is one of the simplest method for transposing rows into columns. *Script 1* shows how a Pivot function can be utilised.

```
SELECT *
FROM
(
   SELECT [PolNumber],
        [PolType],
        [Effective Date],
        [DocName],
        [Submitted]
   FROM [dbo].[InsuranceClaims]
) AS SourceTable PIVOT(AVG([Submitted]) FOR [DocName] IN([Doc A],
                                                [Doc B],
                                                [Doc C],
                                                [Doc D],
                                                [Doc E])) AS PivotTable;
```

Script 1: Transpose data using Pivot function
The results of executing *Script 1* are shown in *Figure 1*, as it can be seen, the output is exactly similar to that of *Table 2*.

Figure 1

Furthermore, as we add more policy numbers in our dataset (i.e. *Pol003*), we are able to automatically retrieve them without making any changes to *Script 1*.

Figure 2

Although, we don't have to alter the script to show additional policies, we unfortunately have to update it if we need to return more columns. This is because the Pivot function works with only a predefined list of possible fields. Thus, in order to return *[Doc F]* column, we would firstly need to update the FOR clause in *Script 1* to include *[Doc F]* and only then would the output reflect *[Doc F]* as shown in *Figure 3*.

Figure 3

However, imagine if business later decides to add 100 more documents that are required to process a claim? It would mean that you need to update your Pivot script and manually add those 100 fields. Thus, although transposing rows using Pivot operator may seem simple, it may later be difficult to maintain.

Performance Breakdown

The actual estimated plan depicted in *Figure 4*, indicates that only a single scan was made against the base table with a majority of the cost (at 77.6%) used for sorting data.

Figure 4

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans
In terms of operational tree, the highest increased in I/O was recorded during the Sort operation at 0.01 milliseconds.

Figure 5

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans

Option #2: CURSOR

Although the general consensus in the professional community is to stay away from SQL Server Cursors, there are still instances whereby the use of cursors is recommended. I suppose if they were totally useless, Microsoft would have deprecated their usage long ago, right? Anyway, Cursors present us with another option to transpose rows into columns. *Script 2* displays a T-SQL code that can be used to transpose rows into columns using the Cursor function.

```
DECLARE @PolNumber NVARCHAR(255), @PolNumber5 NVARCHAR(255),
@PolType VARCHAR(255), @DocName
NVARCHAR(255), @Submitted INT, @Eff DATE, @message_T
NVARCHAR(MAX);
SET @message_T = '';
SET @PolNumber5 = '';
DECLARE policyDocs_csr CURSOR
FOR
   SELECT [PolNumber],
        [PolType],
        [Effective Date],
        [DocName],
        [Submitted]
   FROM [dbo].[InsuranceClaims]
   ORDER BY [PolNumber];
OPEN policyDocs_csr;
FETCH NEXT FROM policyDocs_csr INTO @PolNumber, @PolType, @Eff,
@DocName, @Submitted;
WHILE @@FETCH_STATUS = 0
   BEGIN
      IF @PolNumber5 <> @PolNumber
         SET @message_T = @message_T+CHAR(13)+@PolNumber+' |
'+@PolType+' | '+CONVERT(VARCHAR,
@eff)+' | '+@DocName+' ( '+CONVERT(VARCHAR, isnull(@submitted, ''))+' ) | ';
         ELSE
      IF @PolNumber5 = @PolNumber
         SET @message_T = @message_T+@DocName+' (
'+CONVERT(VARCHAR, isnull(@submitted, ''))+' ) |
';
      SET @PolNumber5 = @PolNumber;
      FETCH NEXT FROM policyDocs_csr INTO @PolNumber, @PolType, @Eff,
@DocName, @Submitted;
   END;
IF @@FETCH_STATUS <> 0
   PRINT @message_T;
CLOSE policyDocs_csr;
DEALLOCATE policyDocs_csr;
```

Script 2: Transpose data using Cursor function

Execution of *Script 2* lead to the result set displayed in *Figure 6* yet, the Cursor option uses far more lines of code than its T-SQL Pivot counterpart.

Figure 6

Similar to the Pivot function, the T-SQL Cursor has the dynamic capability to return more rows as additional policies (i.e. *Pol003*) are added into the dataset, as shown in *Figure 7*:

Figure 7

However, unlike the Pivot function, the T-SQL Cursor is able to expand to include newly added fields (i.e. *[Doc F]*) without having to make changes to the original script.

Figure 9

Performance Breakdown

The major limitation of transposing rows into columns using T-SQL Cursor is a limitation that is linked to cursors in general – they rely on temporary objects, consume memory resources and processes row one at a time which could all result into significant performance costs. Thus, unlike in the Pivot function wherein the majority of the cost was spent sorting the dataset, the majority of cost in the Cursor option is split between the Sort operation (at 46.2%) as well as the temporary *TempDB* object (at 40.5%).

Figure 10

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans
Similar to the operational tree of the Pivot function, the operator with the higher percentages in the execution plan of the Cursor function are likely to consume more I/O resources than other operators. In this case, both the Sort and temporary *TempDB* objects recorded the most I/O usage cost at 0.01 milliseconds each.

Figure 11

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans

Option #3: XML

The XML option to transposing rows into columns is basically an optimal version of the PIVOT in that it addresses the dynamic column limitation. The XML version of the script addresses this limitation by using a combination of XML Path, dynamic T-SQL and some built-in T-SQL functions such as *STUFF* and *QUOTENAME*. The version of the script that uses XML function to transpose rows into columns is shown in *Script 3*.

```
DECLARE @cols NVARCHAR(MAX), @query NVARCHAR(MAX);
SET @cols = STUFF(
                 (
                    SELECT DISTINCT
                          ','+QUOTENAME(c.[DocName])
                    FROM [dbo].[InsuranceClaims] c FOR XML PATH(''), TYPE
                 ).value('.', 'nvarchar(max)'), 1, 1, '');
SET @query = 'SELECT [PolNumber], '+@cols+'from (SELECT [PolNumber],
          [PolType],
          [submitted] AS [amount],
          [DocName] AS [category]
   FROM [dbo].[InsuranceClaims]
   )x pivot (max(amount) for category in ('+@cols+')) p';
EXECUTE (@query);
```

Script 3: Transpose data using XML function
The output of *Script 3* execution is shown in *Figure 12*.

Figure 12

Similar to T-SQL Pivot and Cursor options, newly added policies (i.e. *Pol003*) are retrievable in the XML option without having to update the original script. Furthermore, the XML option is also able to cater for dynamic field names (i.e. *[Doc F]*) as shown in *Figure 13*.

Figure 13

Performance Breakdown

The execution plan of *Script 3* is almost similar to that of the Pivot function script in that majority of the cost is taken up by the *Sort* operator with the *Table scan* being the second most costly operation.

Figure 14

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans
In terms of I/O cost, the Sort operation used the longest time at 0.01 milliseconds.

Figure 15

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans

Option #4: Dynamic SQL

Another alternative to the optimal XML option is to transpose rows into columns using purely dynamic SQL – without XML functions. This option utilises the same built-in T-SQL functions that are used in the XML option version of the script as shown in *Script 4*.

```
DECLARE @columns NVARCHAR(MAX), @sql NVARCHAR(MAX);
SET @columns = N'';
SELECT @columns+=N', p.'+QUOTENAME([Name])
FROM
(
   SELECT [DocName] AS [Name]
   FROM [dbo].[InsuranceClaims] AS p
   GROUP BY [DocName]
) AS x;
SET @sql = N'
SELECT [PolNumber], '+STUFF(@columns, 1, 2, '')+' FROM (
SELECT [PolNumber], [Submitted] AS [Quantity], [DocName] as [Name]
   FROM [dbo].[InsuranceClaims]) AS j PIVOT (SUM(Quantity) FOR [Name] in
  ('+STUFF(REPLACE(@columns, ', p.[', ',['), 1, 1, '')+')) AS p;';
EXEC sp_executesql
```

Script 4: Transpose data using Dynamic SQL function

Again, like all the other options, the script using Dynamic SQL returns data in a correctly transposed layout. Similar to T-SQL Cursor and XML options, Dynamic SQL is able to cater for newly added rows and columns without any prior updates to the script.

Figure 16

Performance Breakdown

Except for using XML functions, the Dynamic SQL option is very similar to the XML option. It is not surprising then that its execution plan and operations tree will look almost similar to that of the XML option.

Figure 17

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans
Figure 18

The screenshot is from ApexSQL Plan, a tool to view and analyze SQL Server query execution plans

Conclusion

In this article, we've had a look at available T-SQL options for transposing rows into columns. The Pivot option was shown to be the simplest option yet its inability to cater for dynamic columns made it the least optimal option. The T-SQL Cursor option addressed some of the limitations of the Pivot option though at a significant cost of resources and SQL Server performance. Finally, the XML and the Dynamic SQL options proved to be the best optimal options in terms of transposing rows into columns with favorable performance results and effective handling dynamic rows and columns.

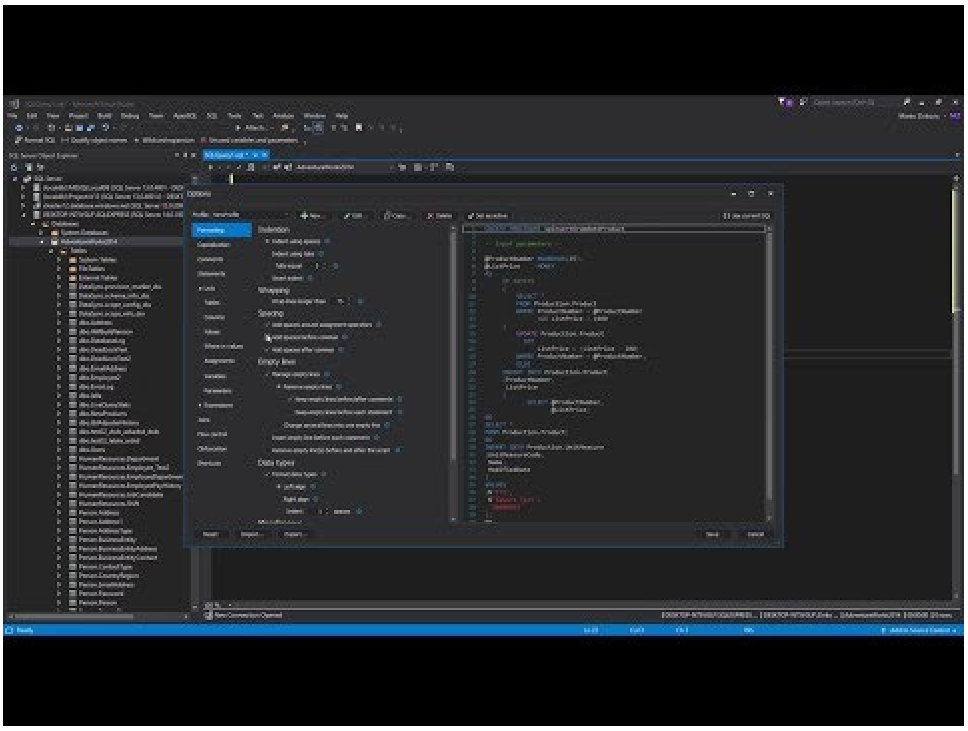
Downloads

Sample Insurance Claims data

References

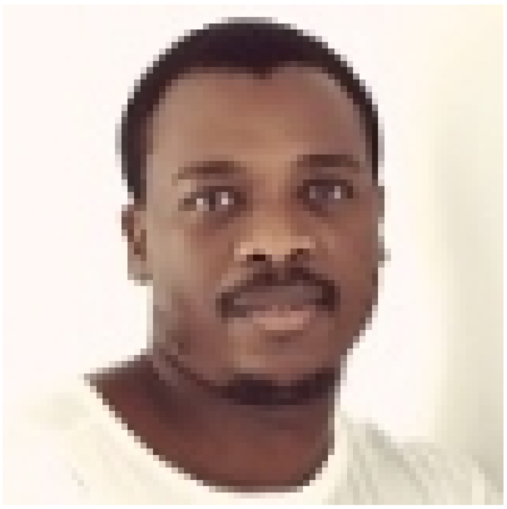
See more

To boost SQL coding productivity, check out these SQL tools for SSMS and Visual Studio including T-SQL formatting, refactoring, auto-complete, text and data search, snippets and auto-replacements, SQL code and object comparison, multi-db script comparison, object decryption and more



Watch Video At: https://youtu.be/_lH7yihq6TA

Sifiso W. Ndlovu



Sifiso is a Johannesburg based certified professional within a wide range of Microsoft Technology Competencies such SQL Server and Visual Studio Application Lifecycle Management.

He is the member of the Johannesburg SQL User Group and also hold a Master's Degree in MCom IT Management from the University of Johannesburg.

He currently works for Sambe Consulting as a Principal Consultant.

View all posts by Sifiso W. Ndlovu