Why Scalar Functions Can Be Costly

By Mickey Stuewe, 2015/12/11

User-defined functions (UDFs) were introduced in SQL Server 2000. I thought that they were the best things since sliced bread. Why? Because I could use them to modularize my code. What I didn't know at the time was how costly they can be to use.

Definitions

User-defined functions can be broken up into three categories:

- **Scalar Functions** always return a single value and are most commonly found in the SELECT clauses and predicates.
- **Multi-line Table Functions** always return a table variable. They can have one to N SQL Statements.
- **In-line Table Functions** always return a table (not a table variable). They can only have one SQL Statement.

In this article, we look at scalar functions.

Scalar Functions

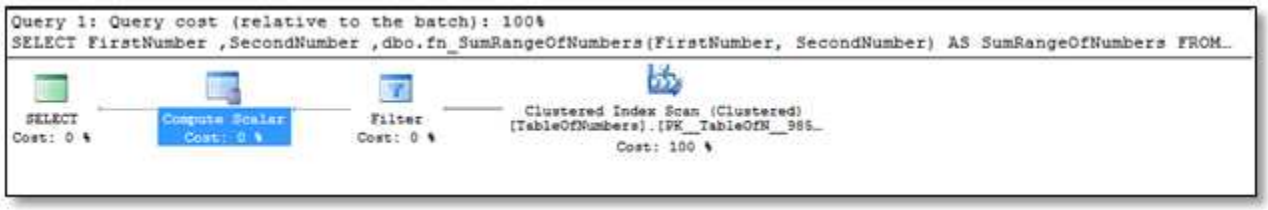
In this article, we'll take a look at how Scalar Functions can be costly, how to identify the cost, and how to rewrite the query that is using the Scalar Function. In future articles, we'll talk about Multi-line and In-line Table Functions.

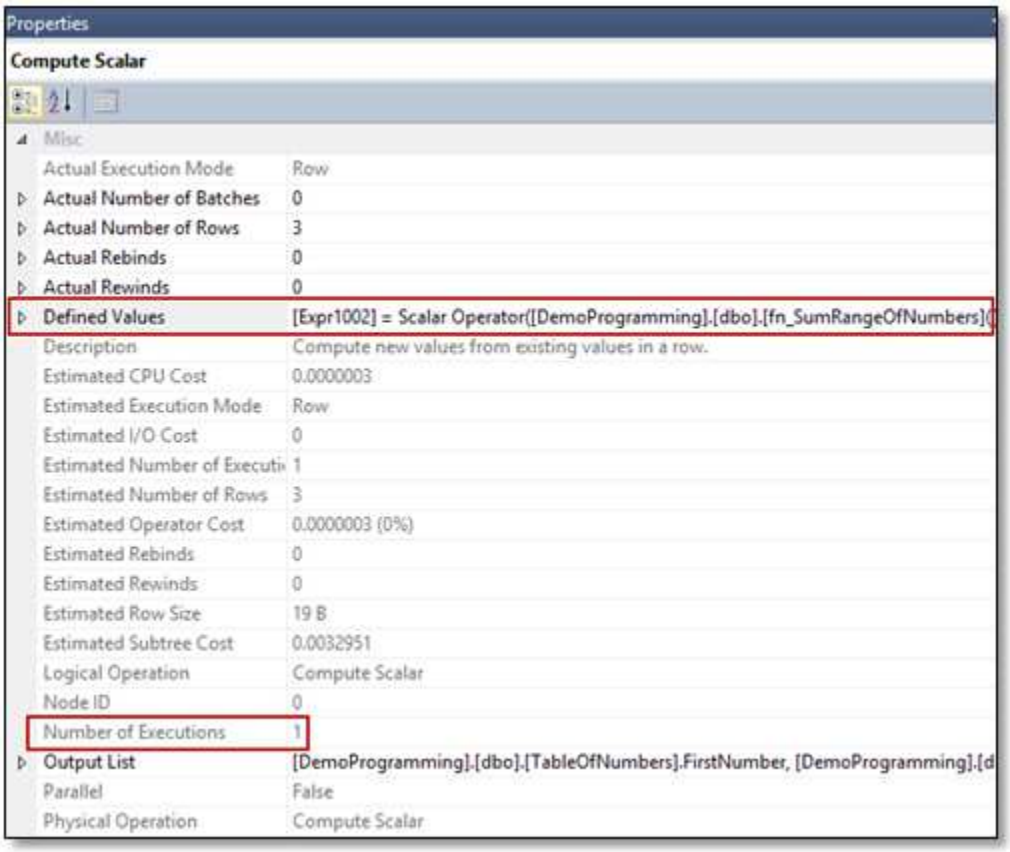
Scalar Functions are costly, due to the fact that they execute multiple times in the SQL Statement it's been used in. If the Scalar Function exists in the SELECT field list, then it executes once for every row returned. If the Scalar Function exists in the predicate, then it executes once for every row being analyzed. Consider the query below where the Scalar Function, dbo.fn_SumRangeOfNumbers() sums a range of numbers based on the two numbers being passed in. The query returns all rows where the sum of the range is less than 1500.

```
SELECT
    FirstNumber
   ,SecondNumber
   ,dbo.fn_SumRangeOfNumbers(FirstNumber, SecondNumber) AS SumRangeOfNumbers
FROM
    dbo.TableOfNumbers
WHERE
    dbo.fn_SumRangeOfNumbers(FirstNumber, SecondNumber) < 1500
```

	FirstNumber	SecondNumber	SumRangeOfNumbers
1	56	58	171
2	1	100	5050
3	11	153	11726
4	54	57	222
5	58	134	7392
6	85	96	1086
7	69	126	5655
8	96	150	6765
9	54	119	5709
10	29	94	4059

In this particular sample, the TableOfNumbers table contains ten rows and the query returns three rows (highlighted above). On the surface, the execution plan shows a simple SELECT statement for the query. The Scalar Function is represented by the Compute Scalar operator. In the property window for the Compute Scalar operator, it states that the Scalar Function was only executed one time. What you can't see is what is "actually" happening behind the scenes. Let's take a look at what is being obfuscated from the Execution Plan.





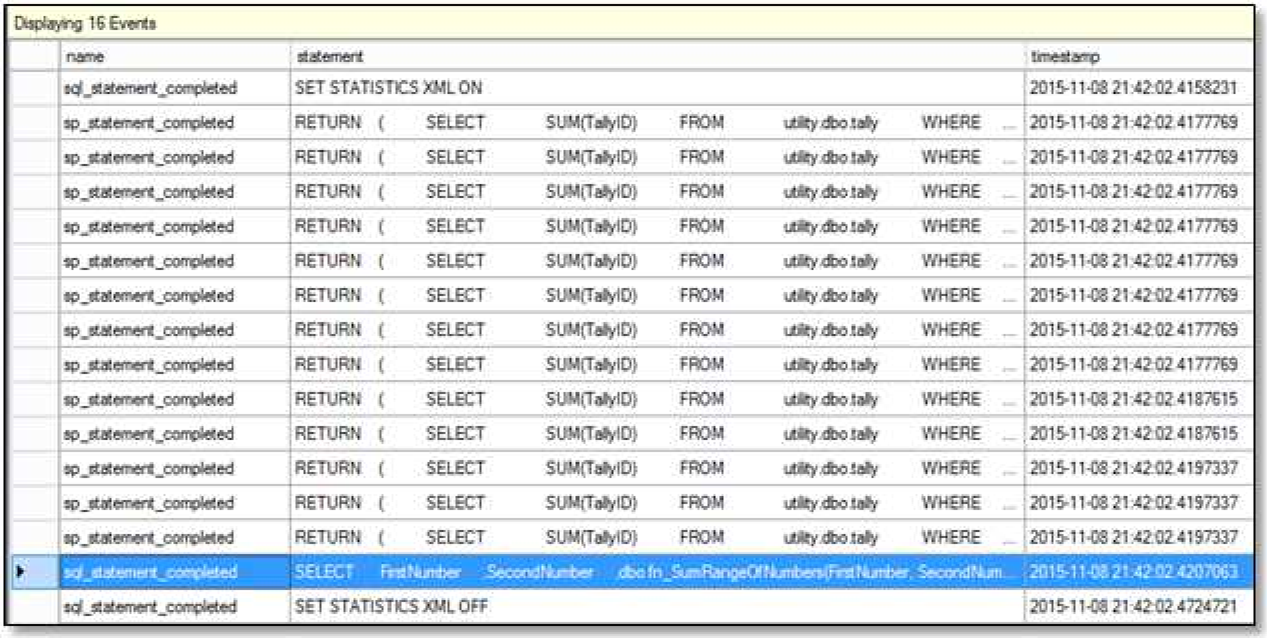
Properties

Compute Scalar

Misc	
Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	3
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[Expr1002] = Scalar Operator([DemoProgramming].[dbo].[fn_SumRangeOfNumbers](
Description	Compute new values from existing values in a row.
Estimated CPU Cost	0.0000003
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executi	1
Estimated Number of Rows	3
Estimated Operator Cost	0.0000003 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	19 B
Estimated Subtree Cost	0.0032951
Logical Operation	Compute Scalar
Node ID	0
Number of Executions	1
Output List	[DemoProgramming].[dbo].[TableOfNumbers].FirstNumber, [DemoProgramming].[d
Parallel	False
Physical Operation	Compute Scalar

Under the Covers

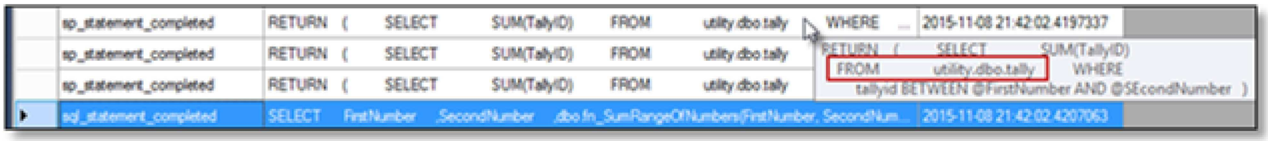
Below I created an Extended Events session that captures each SQL statement that was executed for my query in my session. The blue row highlights the query that I wrote. Each execution of the Scalar Function is represented by the thirteen rows above it. Ten rows are for the Scalar Function in the WHERE statement, which was checking to see if the sum was below 1500 and three rows for the three rows that will be returned in the result set.



Displaying 16 Events

name	statement	timestamp
sql_statement_completed	SET STATISTICS XML ON	2015-11-08 21:42:02.4158231
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4177769
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4187615
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4187615
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4197337
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4197337
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4197337
sql_statement_completed	SELECT FirstNumber ,SecondNumber ,dbo.fn_SumRangeOfNumbers(FirstNumber, SecondNum...	2015-11-08 21:42:02.4207063
sql_statement_completed	SET STATISTICS XML OFF	2015-11-08 21:42:02.4724721

Let’s take a closer look at what is actually going on in the Scalar Function. Below you will see that the scalar function is summing rows from a tally table. So, there are 13 calls to another table that is not represented in the original execution plan. Furthermore, those queries have their own execution plan.



sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally WHERE ...	2015-11-08 21:42:02.4197337
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally	
sp_statement_completed	RETURN (SELECT SUM(TallyID) FROM utility.dbo.tally	
sql_statement_completed	SELECT FirstNumber ,SecondNumber ,dbo.fn_SumRangeOfNumbers(FirstNumber, SecondNum...	2015-11-08 21:42:02.4207063

```
RETURN (    SELECT          SUM(TallyID)
  FROM          utility.dbo.tally          WHERE
    tallyid BETWEEN @FirstNumber AND @SEcondNumber  )
```

Can you imagine what would happen if there were a million rows in the table being used in the query? That would total to 1,300,001 SQL statements being executed for a result set of 300,000 rows with an execution plan showing a simple SELECT statement.

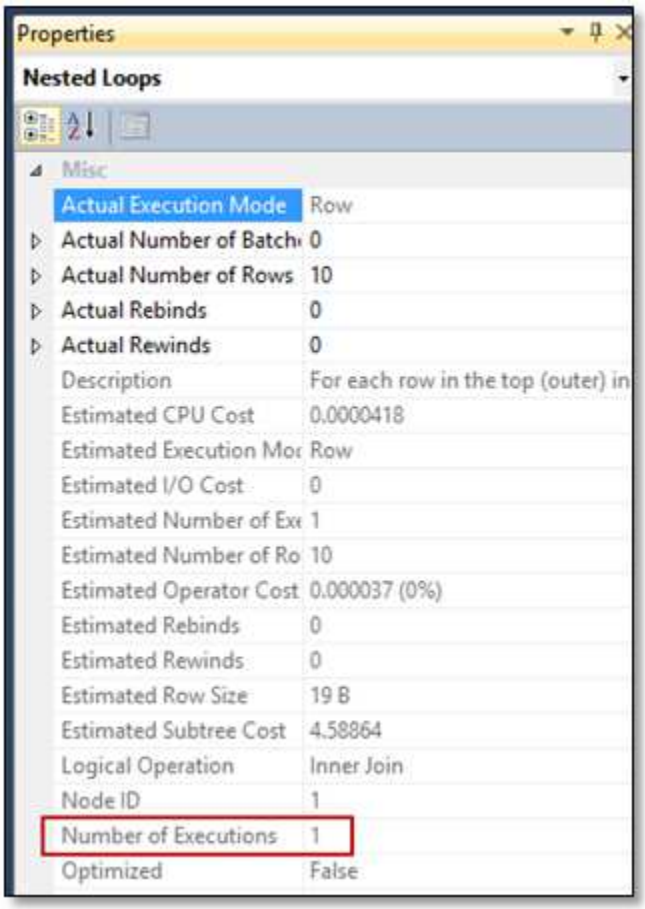
Note: A tally table is a narrow table of sequential numbers. It is also known as a numbers table. It is used to do set based logic in order to help find gaps in dates, Ids, or other numerical data. It can also be used to split delimited strings into a table or array and remove the need for cursors. Jeff Moden (b) wrote a fabulous article on tally tables. He shows how to create one and use it in queries.

How to Rewrite the Query

This query can be rewritten by using the CROSS APPLY join operator with a sub query. This will do two things: it will show that there is a second table being used in the Execution Plan and it will result in one query statement being executed.

```
SELECT
    ton.FirstNumber
   ,ton.SecondNumber
   ,s.SumRangeOfNumbers
FROM
    dbo.TableOfNumbers AS ton
    CROSS APPLY (SELECT
                    SUM(TallyId) AS SumRangeOfNumbers
                 FROM
                    Utility.dbo.tally
                 WHERE
                    TallyID BETWEEN ton.FirstNumber AND ton.SecondNumber
                ) AS s
WHERE
    s.SumRangeOfNumbers < 1500
```

If you look at the Properties of the Nested Loop, you will see that the Number of Executions is 1. You will also see that one SQL Statement was executed and captured during the Extended Events session.



When to Use Scalar Functions

Scalar Functions do have a use. They can be used to perform mathematical calculations or manipulate strings. The key is not to query tables or views in Scalar Functions. That is where the trouble begins.

In the query below, the Scalar Function is verifying that the casing is correct on any name passed to the scalar function, dbo.StandardNameFormat(). I then used the scalar function to correct the casing on the first and last names of my customer table.

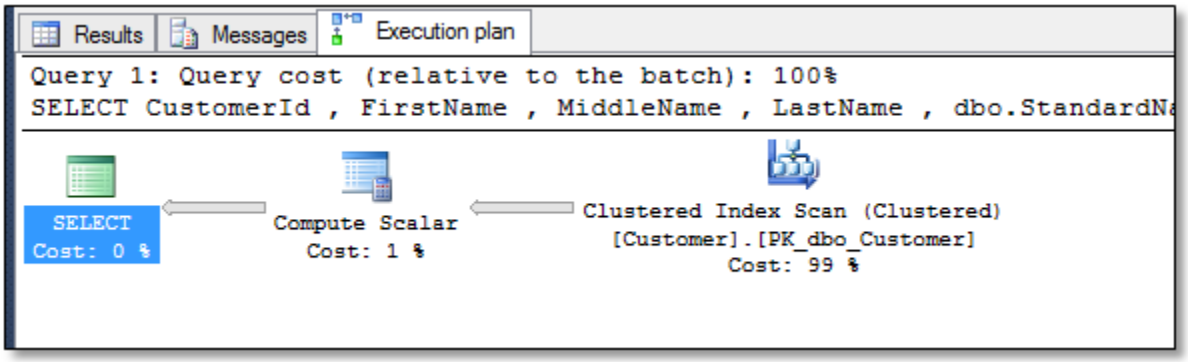
```sql
ALTER FUNCTION dbo.StandardNameFormat ( @Name AS VARCHAR(50) )
RETURNS VARCHAR(50)
AS
    BEGIN
        RETURN
    (SELECT CONCAT(UPPER(LEFT(LOWER(@Name), 1)), RIGHT(LOWER(@Name),LEN(@Name) - 1)));

    END;
GO
```

```sql
SELECT  CustomerId ,
        FirstName ,
        MiddleName ,
        LastName ,
        dbo.StandardNameFormat(FirstName) AS FormatedFirstName ,
        dbo.StandardNameFormat(LastName) AS FormatedLastName
FROM    dbo.Customer;
```

	CustomerId	FirstName	MiddleName	LastName	FormatedFirstName	FormatedLastName
1	291	GUSTAVO	NULL	achong	Gustavo	Achong
2	293	CATHERINE	R.	abel	Catherine	Abel
3	295	KIM	NULL	abercrombie	Kim	Abercrombie
4	297	HUMBERTO	NULL	acevedo	Humberto	Acevedo
5	299	PILAR	NULL	ackerman	Pilar	Ackerman
6	301	FRANCES	B.	adams	Frances	Adams

When I look at the execution plan, I still see the Compute Scalar operator, but when I look at the Extended Event session, I see only one SQL Statement executed. This is a much better use of Scalar Functions.



	name	sql_text	timestamp
	sql_statement_completed	SET STATISTICS XML ON	2015-12-06 18:04:18.8361290
▶	sql_statement_completed	SELECT CustomerId, FirstName, MiddleName, LastName, dbo.StandardNameFormat(FirstName) AS FormatedFirs...	2015-12-06 18:04:19.4038155
	sql_statement_completed	SET STATISTICS XML OFF	2015-12-06 18:04:19.4286643

Conclusion

In this post I demonstrated how costly Scalar Functions can be and how to rewrite them so there aren't multiple queries being run. I also showed how to properly use Scalar Functions. In my next post, I will write about the costs that can occur with using Multi-line Table Functions, and how to rewrite them to perform better.
