



**BRENT OZAR**  
UNLIMITED®

## Why 2 Out of 3 Functions Don't Scale

Let's put the fun in functions!

2.2 p1

# Agenda

## Bad functions:

- Scalar user-defined functions
- Multi-statement table-valued functions
- Functions in table definitions

## Not-so-bad functions:

- SQL Server 2019's Froid technology
- Inline table-valued functions
- CTE and OUTER APPLY

## How to find & fix the bad ones



2.2 p2



**It's not your fault.**

Good developers are taught to use functions.

## Developers are taught to reuse code

Functions are helpful to:

- Encapsulate code and package it for easier reuse
- Make it more test-friendly
- Don't repeat yourself (DRY)

All that stuff is true in app languages,  
and functions don't get a performance overhead there either.



2.2 p4

## Your functions are no good here

- The most important takeaway is that SQL is set-based, and calculating values row by row is painful
- Same applies to cursors, they're almost always performance killers, but that's another issue.
- WHILE loops have similar performance problems, and often get used in functions for splitting strings, padding strings, removing characters, putting names in proper case, etc.



2.2 p5

## Built-in system functions can be bad, too

- Usually (but not always) have bad row estimations
- Which leads to query plans that scan rather than seek
- Can sometimes lead to dramatically higher CPU use

Examples:

- WHERE UPPER(DisplayName) = 'BRENT'
- WHERE LTRIM(RTRIM(DisplayName)) = 'BRENT'
- WHERE ISNULL(DisplayName, '') = 'BRENT'

But in this class, I'm specifically talking about user-defined functions: ones we created.



2.2 p6



## Scalar user-defined functions

## StackOverflow users can earn badges.

Brent Ozar's Stack Overflow profile page. The top navigation bar shows "Database Administrators". The "Activity" tab is selected. On the right, it shows Brent Ozar's profile picture, "Meta User", "Network profile", and "Brent Ozar".

**REPUTATION:** 38,208 (+4.5k) Top 3.39% overall. A chart shows reputation growth from 2017 to 2020.

**BADGES:** 26 (Newest: Publicist), 136, 271. Next badge: Strunk & White (76/80).

**IMPACT:** ~1.3m people reached. 76 posts edited, 13 helpful flags, 659 votes cast.

**Badges:** Recent, Class, Name. 58 Badges listed:

- Publicist × 23
- Booster × 40
- Announcer × 88
- Nice Answer × 89
- Enlightened × 31
- Nice Question × 0
- Lifegadget
- Neonomancer × 13
- Yearling × 5
- Peer Pressure
- Curious
- Self-Learner
- Guru × 3
- Good Question × 2
- Informed
- Suffrage
- Civic Duty
- sql-server-2014
- sql-server
- Enthusiast
- Outspoken
- Explainer
- Student
- Tag Editor
- Excavator
- sql-server
- Organizer
- Mortarboard

**2.2 p8**

## We want to count badges for each user.

Code goals:

- When we show a user's info, we often want to show their # of badges
- Ideally, we code this just once, and it's reusable



2.2 p9

## The scalar function version

```
-- Scalar Function
CREATE FUNCTION dbo.ScalarFunction ( @uid INT )
RETURNS BIGINT
    WITH RETURNS NULL ON NULL INPUT,
         SCHEMABINDING
AS
BEGIN
    DECLARE @BCount BIGINT;
    SELECT @BCount = COUNT_BIG(*)
    FROM dbo.Badges AS b
    WHERE b.UserId = @uid
    GROUP BY b.UserId;
    RETURN @BCount;
END;
```



2.2 p10

## How you call it

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u
```



2.2 p11

**It takes ~8 minutes to return 1,000 rows.**

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
GO
```

150 % 4 Results Messages Execution plan

	DisplayName	(No column name)
1	Community	1
2	Jeff Atwood	328
3	Geoff Dalgas	90
4	Jarrod Dixon	130
5	Joel Spolsky	185
6	Jon Galloway	319
7	Eggs McLaren	24
8	Kevin Dente	87
9	Sneakers O'Toole	3

Query executed successfully. SQL2019 (15.0 RTM) SQL2019\Administrator ... StackOverflow 00:07:48 1,000 rows



2.2 p12

## When a query is this slow, you wanna see what tables it read from.

But SET STATISTICS IO ON completely lies:  
it doesn't show anything that scalar functions do.

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
```

SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:  
CPU time = 0 ms, elapsed time = 0 ms.  
SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 2 ms.

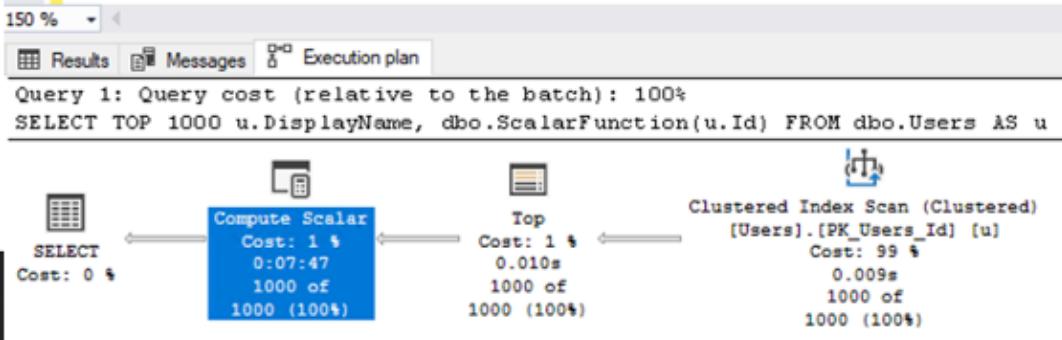
(1000 rows affected)  
Table 'Users'. Scan count 1, logical reads 69, physical reads 0, |  
(1 row affected)

SQL Server Execution Times:  
CPU time = 1861215 ms, elapsed time = 467866 ms.  
SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 0 ms.

2.2 p13

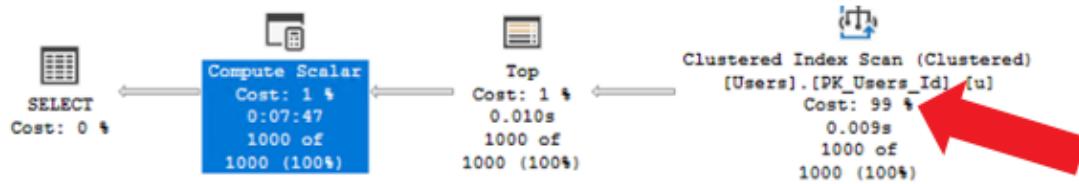
## What about the query plan? It gets worse.

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
GO
```



2.2 p14

## What about the query plan? It gets worse.



SQL Server implies the clustered index scan is 99% of the cost, and that the Compute Scalar was just 1%.

That's because the estimated costs for scalars is completely made up.



2.2 p15

```

SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
GO

```

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT TOP 1000 u.DisplayName, dbo.ScalarFunction(u.Id) FROM dbo.Users AS u

Physical Operation	Logical Operation
Compute Scalar	Compute Scalar
Logical Operation	Logical Operation
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	1000
Actual Number of Batches	0
Estimated I/O Cost	0
Estimated Operator Cost	0.0001 (1%)
Estimated Subtree Cost	0.0162065
Estimated CPU Cost	0.0001
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	1000
Estimated Row Size	598
Actual Rebinds	0
Actual Rewinds	0
Node ID	0

Query executed successfully.

Output List [StackOverflow].[dbo].[Users].DisplayName, Expr1001

Nick was late for beer

The Estimated I/O Cost = 0

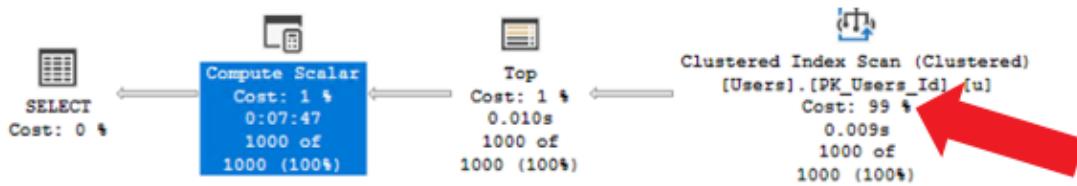
Estimated CPU Cost = 0.0001

These are purely made-up numbers that have nothing to do with the contents of the function.

Microsoft just didn't get around to coding this part of the engine.

2.2 p16

## What about the query plan? It gets worse.



SQL Server implies the clustered index scan is 99% of the cost, and that the Compute Scalar was just 1%.

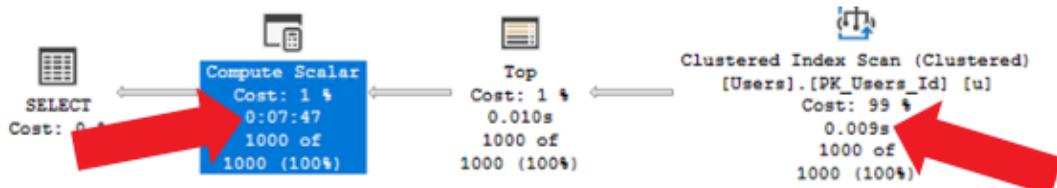
That's because the estimated costs for scalars is completely made up.



2.2 p17

## You have to do some detective work.

If you look closely at the completion times for each operator:



The table scan finished 0.009 seconds into the query.

The Compute Scalar finished 7 minutes, 47 seconds in.



2.2 p18

## Another way to spot it

Look at the properties of the SELECT operator on newer versions:

The screenshot shows a SQL query in the results pane:

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
```

The execution plan shows three operators: a Compute Scalar operator, a Top operator, and a Clustered Index Scan operator.

A red arrow points to the properties grid on the right side of the interface, specifically highlighting the "QueryTimeStats" section. This section contains properties like UdfCpuTime and UdfElapsedTime, which are highlighted in blue.

Property	Value
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0162865
MemoryGrantInfo	
NonParallelPlanReason	CouldNotGenerateValidParallelPlan
Optimization Level	TRIVIAL
OptimizerHardwareDependentProperties	
ParentObjectId	0
QueryHash	0xD07CEC4BEEF3112C
QueryPlanHash	0xEAF1090155A0217E
QueryTimeStats	
CpuTime	1862424
ElapsedTime	467820
UdfCpuTime	1862407
UdfElapsedTime	467802
RetrievedFromCache	true
SecurityPolicyApplied	false
Set Options	ANSI NULLS, ANSI_PADDING

UdfCpuTime, UdfElapsedTime show how much time was spent inside user-defined functions.



2.2 p19

## But my favorite way is sp\_BlitzCache.

sp_BlitzCache						
	Database	Cost	Query Text	Query Type	Warnings	# Executions
1	StackOverflow	0.0162865	SELECT TOP 1000 u.DisplayName, dbo.Scalarfu...	Statement	Long Running Query, Downlevel CE, Forced Serialization, Plan created last 4hrs, Low Cost, High CPU	1
2	StackOverflow	135.252	CREATE FUNCTION dbo.ScalarFunction (@ud ...	Procedure or Function	[dbo].[ScalarFunction] Missing Indexes (1), Parallel, Parameter Sniffing, Downlevel CE, Plan created last 4hrs	1000
3	StackOverflow	135.252	SELECT @BCount =COUNT_BIG() FROM ...	Statement [parent [dbo].[ScalarFunction]]	Missing Indexes (1), Parallel, Parameter Sniffing, Downlevel CE, Plan created last 4hrs	1000

#1 query is the SELECT TOP 1000, which calls the function.

The SELECT's CPU & duration *includes the function's work*.

Now look at the #2 query: it's the function.

The function's CPU time and duration is almost the entire time that the SELECT query worked. That's your big sign.



2.2 p20

## Scalar round up

### The bad:

- Runs once per row
- Cost isn't added to the total query cost
- No info from STATISTICS IO, not much in the plan
- Inhibits parallelism, too: [BrentOzar.com/go/serialudf](http://BrentOzar.com/go/serialudf)

### The good:

This section is intentionally left blank



2.2 p21



## **Multi-statement table-valued user-defined functions (MSTVFs)**

## **MSTVF<sub>s</sub> use table variables to store data.**

Which means all modifications are serial, and

Your estimates will be really low (unless you recompile), and

You won't get column statistics (so SQL won't know how joins will go)



2.2 p23

## How you build it

```
-- Multi Statement Table Valued Function
CREATE FUNCTION dbo.MultiStatementTVF ( @uid INT )
RETURNS @Out TABLE ( BadgeCount BIGINT )
    WITH SCHEMABINDING
AS
BEGIN
    INSERT INTO @Out(BadgeCount)
    SELECT COUNT_BIG(*) AS BadgeCount
    FROM dbo.Badges AS b
    WHERE b.UserId = @uid
    GROUP BY b.UserId;
    RETURN;
END;
```



2.2 p24

## How you call it

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi
```



2.2 p25

Now up to 26.5 minutes.

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

Execution plan details:

- Top**: Cost: 1 %, 0:26:30, 1000 of, 1000 (100%)
- Nested Loops (Inner Join)**: Cost: 47 %, 0:26:30, 1000 of, 1000 (100%)
- Clustered Index Scan (Clustered)**: Cost: 89 %, 0.005s, 1001 of, 10 (10010%)
- Table Valued Function (MultiStatementTVF) [mi]**: Cost: 13 %, 0:26:30, 1000 of, 1100 (90%)

Query executed successfully. SQL2019 (15.0 RTM) | SQL2019\Administrator ... | StackOverflow | 00:26:30 | 1,000 rows

2.2 p26

Now up to 26.5 minutes.



SELECT TOP 1000  
u.DisplayName,  
mi.BadgeCount  
FROM dbo.Users AS u  
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;

100% 4

Results Messages Execution Plan

Query 1: Query cost (reduced): 0 ms

SELECT TOP 1000 u.DisplayName  
Cost: 0 ms

Top Cost: 1 ms  
0:26:30  
1000 of  
1000 (100%)

Query executed successfully. SQL2019 (15.0 RTM) | SQL2019\Administrator ... | StackOverflow | 00:26:30 | 1,000 rows

2.2 p27

## What's STATISTICS IO say this time?

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:  
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server parse and compile time:  
CPU time = 3 ms, elapsed time = 3 ms.

(1000 rows affected)

Table '#B1482DC7'. Scan count 1001, logical reads 1001, physical reads 0,  
Table 'Users'. Scan count 1, logical reads 69, physical reads 0, page ser

(1 row affected)

Query executed successfully.

2.2 p28

## What's STATISTICS IO say this time?



```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

Table '#B1482DC7'. Scan count 1001, logical reads 1001, physical reads 0,  
Table 'Users'. Scan count 1, logical reads 69, physical reads 0, page ser  
(1 row affected)

Query executed successfully. | SQL2019 (15.0 RTM) | SQL2019\Administrator ... | StackOverflow | 00:26:30 | 1,000 rows

2.2 p29

## UdfCpuTime or UdfElapsedTime? No, none.

The screenshot shows the SQL Server Management Studio interface with the following details:

**Query:**

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

**Execution Plan:**

Query 1: Query cost (relative to the batch): 1004

```
SELECT TOP 1000 u.DisplayName, mi.BadgeCount FROM dbo.Users AS u CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

The execution plan shows a Nested Loops (Inner Join) operator. The left side of the join (the outer query) has a cost of 1.4% (0:26:30). The right side (the TVF) has a cost of 47.4% (0:26:30). The total cost is 48.8% (0:26:30), which matches the overall query cost of 1004.

**Query Statistics:**

Statistic	Value
DatabaseContextSettingsId	2
Degree of Parallelism	1
Estimated Number of Rows	1000
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0086923
MemoryGrantInfo	
Optimization Level	FULL
OptimizerHardwareDependentProperties	
ParentObjectId	0
QueryHash	0x0C9CDAC7E65B4975
QueryPlanHash	0xC250F57D7A88299E
QueryTimeStats	
CpuTime	1590539
ElapsedTime	1590577
Reason For Early Termination Of State	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	false
Set Options	ANSI_NULLS: True, ANSI_PADDING: True, ANSI_WARNINGS: True, ARITHABORT: Off, AUTO_CLOSE: Off, AUTO_SHRINK: Off, AUTO_UPDATE_STATISTICS: On, CURSOR_CLOSE_ON_COMMIT: Off, CURSOR_DEFAULT: LOCAL, CURSOR_TIMEOUT: 30, IMPLICIT_TRANSACTIONS: Off, QUOTED_IDENTIFIER: On, RECOMPILE: Off, STATISTICS_NORESET: Off, STATISTICS_SET: Off, TRAILING_SPACE: Off
Statement	SELECT TOP 1000 u.DisplayName, mi.BadgeCount
StatementParameterizationType	0
StatementSqlHandle	0x0900B2F33A8CCFD02046666B065D99ED70
WaitStats	

**Message Bar:**

Query executed successfully.

**Bottom Right:**

2.2 p30

## UdfCpuTime or UdfElapsedTime? No, none.

The screenshot shows a SQL Server Management Studio window with the following details:

- Query:**

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```
- Execution Plan:** A Nested Loops (Inner Join) plan with a cost of 47.4%.
- Results:** A table with 1,000 rows.
- Messages:** "Query executed successfully."
- Properties:** A large pane on the right showing various properties for the query, including:
  - DatabaseContextSettingsId: 2
  - Degree of Parallelism: 1
  - Estimated Number of Rows: 1000
  - Estimated Operator Cost: 0 (0%)
  - Estimated Subtree Cost: 0.0086923
  - MemoryGrantInfo: Optimization Level FULL
  - OptimizerHardwareDependentProperties: ParentObjectId: 0
  - QueryHash: 0x0C9CDAC7E65B4975
  - QueryPlanHash: 0xC258F57D7A88298E
  - QueryTimeStats: CpuTime: 1590539, ElapsedTime: 1590577, Reason For Early Termination Of State: Good Enough Plan Found, RetrievedFromCache: true, SecurityPolicyApplied: False
  - Set Options: ANSI\_NULLS: True, ANSI\_PADDING: True, ANSI\_WARNINGS: True, ARITHABORT: On, AUTO\_CLOSE: Off, AUTO\_SHRINK: Off, AUTO\_UPDATE\_STATISTICS: On, CURSOR\_CLOSE\_ON\_COMMIT: Off, CURSOR\_DEFAULT: LOCAL, CURSOR\_TIMEOUT: 30, IMPLICIT\_TRANSACTIONS: Off, QUOTED\_IDENTIFIER: On, RECOMPILE: Off, STATISTICS\_NORESET: Off, STATISTICS\_SET: Off, TRANSACTION\_ISOLATION LEVEL: READ COMMITTED, TRANSACTION\_SNAPSHOT: Off, TRANSACTION\_XACT\_ABORT: Off, USE\_SNAPSHOT\_ISOLATION: Off
  - Statement: SELECT TOP 1000 u.DisplayName, mi.BadgeCount
  - StatementParameterizationType: 0
  - StatementSqlHandle: 0x0900B2F33A8CCFD02046666B065D99ED70
  - WaitStats
- Background Image:** A photo of Ron Burgundy from the TV show "Anchorman".
- Bottom Bar:** Shows the version (SQL2019 (15.0 RTM)), session (SQL2019\Administrator ...), stack overflow link, time (00:26:30), and rows (1,000 rows). It also features a thumbs-up icon with a heart and the text "2.2 p31".

## But the best part...

The screenshot shows a SQL Server Management Studio window with the following details:

**Query:**

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi;
```

**Execution Plan:**

The execution plan is a Nested Loops (Inner Join) plan. It starts with a **Top** operator (Cost: 1 %) which filters 1000 rows from the **Users** table (Cost: 0 %). This is followed by a **Nested Loops (Inner Join)** operator (Cost: 47 %) which joins the result with the **MultiStatementTVF** function. The **MultiStatementTVF** function is implemented as a **Table Valued Function (MultiStatementTVF) [mi]** (Cost: 13 %). The final step is a **Clustered Index Scan (Clustered)** on the **[Users].[PK\_Users\_Id] [u]** table (Cost: 39 %).

**Results:**

Query 1: Query cost (relative to the batch): 100%

```
SELECT TOP 1000 u.DisplayName, mi.BadgeCount FROM dbo.Users AS u CROSS APPLY dbo.MultiSta...
```

**Messages:**

Query executed successfully.

**Execution plan:**

SQL2019 (15.0 RTM) | SQL2019\Administrator ... | StackOverflow | 00:26:30 | 1,000 rows

**Query Statistics:**

DatabaseContextSettingsId	2
Degree of Parallelism	1
Estimated Number of Rows	1000
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0086923
MemoryGrantInfo	
Optimization Level	FULL
OptimizerHardwareDependentProperties	
ParentObjectId	0
QueryHash	0x0C9CDAC7E65B4975
QueryPlanHash	0xC258F57D7A88298E
QueryTimeStats	
CpuTime	1590539
ElapsedTime	1590577
Reason For Early Termination Of Statement Optimization	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	False
Set Options	ANSI_NULLS: True, ANSI_PADDING: True, ANSI_WARNINGS: True, ARITHABORT: Off, AUTO_CLOSE: Off, AUTO_SHRINK: Off, AUTO_UPDATE_STATISTICS: On, CURSOR_CLOSE_ON_COMMIT: Off, CURSOR_DEFAULT: LOCAL, CURSOR_TIMEOUT: 30, IMPLICIT_TRANSACTIONS: Off, QUOTED_IDENTIFIER: On, RECOMPILE: Off, STATISTICS_NORESET: Off, STATISTICS_SET: Off, TRANSACTION_ISOLATION LEVEL: READ COMMITTED, XACT_ABORT: On
Statement	SELECT TOP 1000 u.DisplayName, mi.BadgeCount FROM dbo.Users AS u CROSS APPLY dbo.MultiSta...
StatementParameterizationType	0
StatementSqlHandle	0x0900B2F33A8CCFD020466
WaitStats	

**Reason For Early Termination Of Statement Optimization:**

Reason For Early Termination Of Statement Optimization

/F(u.Id) AS mi;	Estimated Subtree Cost	0.0086923
0%	MemoryGrantInfo	
ON dbo.Users AS u CROSS APPLY dbo.MultiSta...	Optimization Level	FULL
	OptimizerHardwareDependentProperties	
- [Users].[PK_Users_Id] [u]	ParentObjectId	0
Cost: 39 %	QueryHash	0x0C9CDAC7E65B4975
0.005s	QueryPlanHash	0xC258F57D7A88298E
1001 of	QueryTimeStats	
10 (10010%)	CpuTime	1590539
	Elapsed Time	1590577
- Table Valued Function	Reason For Early Termination Of State	Good Enough Plan Found
[MultiStatementTVF] [mi]	RetrievedFromCache	true
Cost: 13 %	SecurityPolicyApplied	False
0:26:30	Set Options	ANSI_NULLS: True, ANSI_PA
1000 of	Statement	SELECT TOP 1000 u.DisplayNa
1100 (90%)	StatementParameterizationType	0
	StatementSqlHandle	0x0900B2F33A8CCFD020466
	WaitStats	
<b>Reason For Early Termination Of Statement Optimization</b>		
Reason For Early Termination Of Statement Optimization		

( SQL2019\Administrator ... | StackOverflow | 00:26:30 | 1,000 rows )

ElapsedTime	1590577
Reason For Early Termination Of State	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	False
+ Set Options	ANSI_NULLS: True, ANSI_PADDING: True, ANSI_WARNINGS: True, ARITHABORT: Off, AUTO_CLOSE: Off, AUTO_SHRINK: Off, AUTO_UPDATE_STATISTICS: On, CURSOR_CLOSE_ON_COMMIT: Off, CURSOR_DEFAULT: Local, CURSOR_TIMEOUT: 30, IMPLICIT_TRANSACTIONS: Off, QUOTED_IDENTIFIER: On, RECOMPILE: Off, STATISTICS_NORECOMPUTE: Off, USE_SNAPSHOT_ISOLATION: Off, USE_TRANSACTION_LOGGING: Off, USE_TORN_PAGE_DETECTION: Off, USE_TUTLEDGE: Off, USE_XACT_ABORT: Off, XML: Off
Statement	SELECT TOP 1000 u.DisplayName, u.SalesOrderCount, u.SalesOrderDetailCount, u.SalesOrderHeaderCount, u.SalesOrderLineCount, u.SalesOrderNumber, u.SalesOrderStatus, u.SalesOrderType, u.SalesTerritoryID, u.SalesTerritoryName, u.SalesYTD, u.TerritoryID, u.TerritoryName, u.TotalSales FROM SalesLT.Customer AS u ORDER BY u.SalesTerritoryName
StatementParameterizationType	0
StatementSqlHandle	0x0900B2F33A8CCFD020466
+ WaitStats	

ElapsedTime	1590577
Reason For Early Termination Of State	Good Enough Plan Found
	<code>_NULLS: True, ANSI_PA CT TOP 1000 u.DisplayL 00B2F33A8CCFD020466</code>
	

## MSTVF round up

### The bad:

- Table variable guarantees at least a serial zone in the plan
- Takes several times longer than a scalar
- Some information in the execution plan, but misleading
- Some information in STATISTICS IO, but also misleading

### The good:

- They get just a little better in SQL Server 2017.



2.2 p36



**It gets worse.**

Functions in table definitions are horrible.

## Scalar functions can be used in more places

- Computed columns
- Check constraints

Using them in either of these makes

**ALL QUERIES AGAINST THE TABLE SERIAL**

This includes maintenance like index rebuilds, CHECKDB.



## Blogs to prove it

### Still Serial After All These Years:

<https://www.brentozar.com/archive/2016/01/still-serial-after-all-these-years/>

### Scalar Functions In Computed Columns:

<https://www.brentozar.com/archive/2016/01/another-reason-why-scalar-functions-in-computed-columns-is-a-bad-idea/>

### Scalar Functions In Check Constraints:

<https://www.brentozar.com/archive/2016/04/another-hidden-parallelism-killer-scalar-udfs-check-constraints/>



2.2 p39



# Let's get some relief.

SQL Server 2019, or manually rewrite 'em to go inline.

## After a decade, Microsoft saw the light.

When Microsoft started hosting your databases in Azure SQL DB, they suddenly realized how CPU-intensive functions were. Go figure.

SQL Server 2019 can inline some scalar functions:

<https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining>

There are a lot of limitations, so not all functions will inline:

- Can't call time-dependent functions like GETDATE()
- The UDF can't have table variables or TVPs
- The UDF can't be referenced in a GROUP BY or ORDER BY
- The UDF can't be used in a computed column, check constraint, or partitioned function



2.2 p41

# This stuff is amazeballs complicated

White paper about the internals: <https://arxiv.org/pdf/1712.00498>

Possibly extensible to C#, Java, R, Python hosted in SQL Server

Optimization of Imperative Programs in a Relational Database<sup>\*</sup>

Technical Report

Karthik Ramachandra  
Microsoft Gray Systems Lab  
karam@microsoft.com

Kwanghyun Park  
Microsoft Gray Systems Lab  
kwpark@microsoft.com

César Galindo-Legaria  
Microsoft  
cesarg@microsoft.com

K. Venkatesh Emani<sup>†</sup>  
IIT Bombay  
venkateshe@csail.its.iitb.ac.in

Conor Cunningham  
Microsoft  
conorc@microsoft.com

**ABSTRACT**  
For decades, RDBMSs have supported declarative SQL as well as imperative functions and procedures as ways for users to write programs in various languages (such as Transact-SQL, C#, Java and R) using imperative constructs such as variable assignments, conditional branching, and loops.

Aug 2019

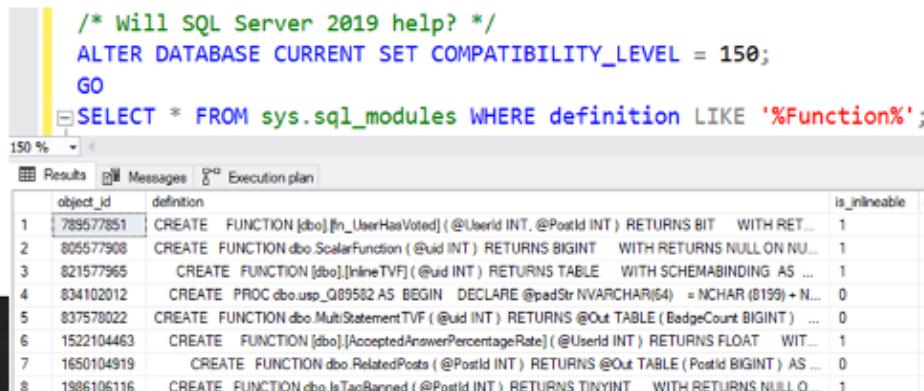
2.2 p42

## Will your functions be a good fit?

Set up a SQL Server 2019 server, restore your databases into it

Check the `is_inlineable` column in `sys.sql_modules`:

That doesn't mean it'll actually get inlined:  
the decision is made at compile time, based on the query calling the UDF



object_id	definition	is_inlineable	u
1	CREATE FUNCTION [dbo].[fn_UserHasVoted] ( @UserId INT, @PostId INT ) RETURNS BIT WITH RET... GO	1	1
2	CREATE FUNCTION dbo.ScalarFunction ( @uid INT ) RETURNS BIGINT WITH RETURNS NULL ON NU... 3	1	1
3	CREATE FUNCTION [dbo].[InlineTVF] ( @uid INT ) RETURNS TABLE WITH SCHEMABINDING AS ... 4	1	1
4	CREATE PROC dbo.usp_Q89582 AS BEGIN DECLARE @adStr NVARCHAR(64) = NCHAR(8199)+N... 5	0	1
5	CREATE FUNCTION dbo.MultiStatementTVF ( @uid INT ) RETURNS @Out TABLE ( BadgeCount BIGINT ) ... 6	0	1
6	CREATE FUNCTION [dbo].[AcceptedAnswerPercentageRate] ( @UserId INT ) RETURNS FLOAT WIT... 7	1	1
7	CREATE FUNCTION dbo.RelatedPosts ( @PostId INT ) RETURNS @Out TABLE ( PostId BIGINT ) AS ... 8	0	1
8	CREATE FUNCTION dbo.IsTopicBanned ( @PostId INT ) RETURNS TINYINT WITH RETURNS NULL O... 9	1	1

2.2 p43

## There are a huge list of limitations

<https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining>

A scalar T-SQL UDF can be inlined if all of the following conditions are true:

- The UDF is written using the following constructs:
  - `DECLARE`, `SET`: Variable declaration and assignments.
  - `SELECT`: SQL query with single/multiple variable assignments <sup>1</sup>.
  - `IF` / `ELSE`: Branching with arbitrary levels of nesting.
  - `RETURN`: Single or multiple return statements.
  - `UDF`: Nested/recursive function calls <sup>2</sup>.
  - Others: Relational operations such as `EXISTS`, `ISNULL`.
- The UDF does not invoke any intrinsic function that is either time-dependent (such as `GETDATE()`) or has side effects <sup>3</sup> (such as `NEWSEQUENTIALID()`).

- The UDF does not reference table variables or table-valued parameters.
- The query invoking a scalar UDF does not reference a scalar UDF call in its GROUP BY clause.
- The query invoking a scalar UDF in its select list with DISTINCT clause does not have ORDER BY clause.
- The UDF is not used in ORDER BY clause.
- The UDF is not natively compiled (interop is supported).
- The UDF is not used in a computed column or a check constraint definition.
- The UDF does not reference user-defined types.
- There are no signatures added to the UDF.
- The UDF is not a partition function.
- The UDF does not contain references to Common Table Expressions (CTEs).
- The UDF does not contain references to intrinsic functions that may alter the results when inlined (such as @@ROWCOUNT)<sup>4</sup>.
- The UDF does not contain aggregate functions being passed as parameters to a scalar UDF<sup>4</sup>.
- The UDF does not reference built-in views (such as OBJECT\_ID)<sup>4</sup>.
- The UDF does not reference XML methods<sup>5</sup>.
- The UDF does not contain a SELECT with ORDER BY without a TOP 1 clause<sup>5</sup>.
- The UDF does not contain a SELECT query that performs an assignment in conjunction with the ORDER BY clause (such as SELECT @x = @x + 1 FROM table1 ORDER BY col1)<sup>5</sup>.
- The UDF does not contain multiple RETURN statements<sup>6</sup>.
- The UDF is not called from a RETURN statement<sup>6</sup>.
- The UDF does not reference the STRING\_AGG function<sup>6</sup>.
- The UDF does not reference remote tables<sup>7</sup>.
- The UDF-calling query does not use GROUPING SETS, CUBE, or ROLLUP<sup>7</sup>.
- The UDF-calling query does not contain a variable that is used as a UDF parameter for assignment (for example, SELECT @y = 2, @x = UDF(@y))<sup>7</sup>.

2.2 p45

## Microsoft keeps adding more limitations

Since SQL Server 2019 released, most of the Cumulative Updates have removed inline function support for more and more functions:

<https://support.microsoft.com/en-us/help/4538581/fix-scalar-udf-inlining-issues-in-sql-server-2019>



### FIX: Scalar UDF Inlining issues in SQL Server 2019

Applies to: SQL Server 2019 on Linux, SQL Server 2019 on Windows

#### Symptoms

User-Defined Functions (UDFs) that are implemented in Transact-SQL and that return a single data value are referred to as T-SQL Scalar User-Defined Functions (UDFs).

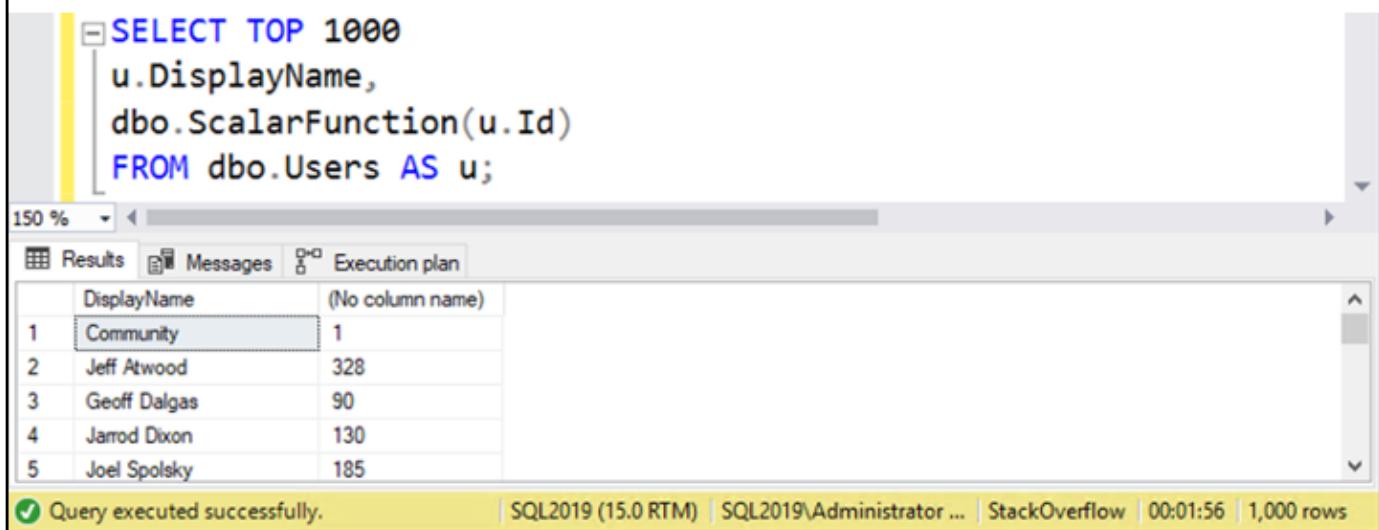
46

This cumulative update also **blocks** inlining in the following scenarios:

- If the UDF references certain intrinsic functions (for example, @@ROWCOUNT) that may alter the results when inlined (added in Microsoft SQL Server 2019 CU2)
- When aggregate functions are passed as parameters to a scalar UDF (added in Microsoft SQL Server 2019 CU2)
- If the UDF references built-in views (for example: OBJECT\_ID) (added in Microsoft SQL Server 2019 CU2)
- If the UDF uses XML methods (added in Microsoft SQL Server 2019 CU4)
- If the UDF contains a SELECT with ORDER BY without a "TOP 1" (added in Microsoft SQL Server 2019 CU4)
- If the SELECT query performs an assignment in conjunction with the ORDER BY clause (for example, SELECT @x = @x + 1 FROM table ORDER BY *column\_name*) (*added in Microsoft SQL Server 2019 CU4*)
- If the UDF contains multiple RETURN statements (added in Microsoft SQL Server 2019 CU5)
- If the UDF is called from a RETURN statement (added in Microsoft SQL Server 2019 CU5)
- If the UDF references the STRING\_AGG function (added in Microsoft SQL Server 2019 CU5)
- If the UDF definition references remote tables (added in Microsoft SQL Server 2019 CU6)
- If the UDF-calling query uses GROUPING SETS, CUBE, or ROLLUP (added in Microsoft SQL Server 2019 CU6)
- If the UDF-calling query contains a variable that is used as a UDF parameter for assignment (for example, SELECT @y=2, @x=UDF(@y)) (added in Microsoft SQL Server 2019 CU6)

## Let's try our ScalarFunction example

It used to take ~8 minutes. In 2019 compat level: 2 minutes!



A screenshot of the SQL Server Management Studio (SSMS) interface. The query window displays a T-SQL SELECT statement. The results pane shows a table with two columns: 'DisplayName' and '(No column name)'. The data consists of five rows with the following values:

	DisplayName	(No column name)
1	Community	1
2	Jeff Atwood	328
3	Geoff Dalgas	90
4	Jarrod Dixon	130
5	Joel Spolsky	185

The status bar at the bottom indicates "Query executed successfully." and provides session details: SQL2019 (15.0 RTM), SQL2019\Administrator, StackOverflow, 00:01:56, and 1,000 rows.

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
FROM dbo.Users AS u;
```

## STATISTICS IO now shows what's going on

```
SELECT TOP 1000
    u.DisplayName,
    dbo.ScalarFunction(u.Id)
    FROM dbo.Users AS u;
GO

150 % - <

Results Messages Execution plan

SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
    CPU time = 4 ms, elapsed time = 4 ms.

(1000 rows affected)
Table 'Worktable'. Scan count 1000, logical reads 83182561, physical re
Table 'Badges'. Scan count 1, logical reads 168260, physical reads 0, p
Table 'Users'. Scan count 1, logical reads 69, physical reads 0, page s

(1 row affected)

SQL Server Execution Times:
    CPU time = 116157 ms, elapsed time = 116255 ms.
```

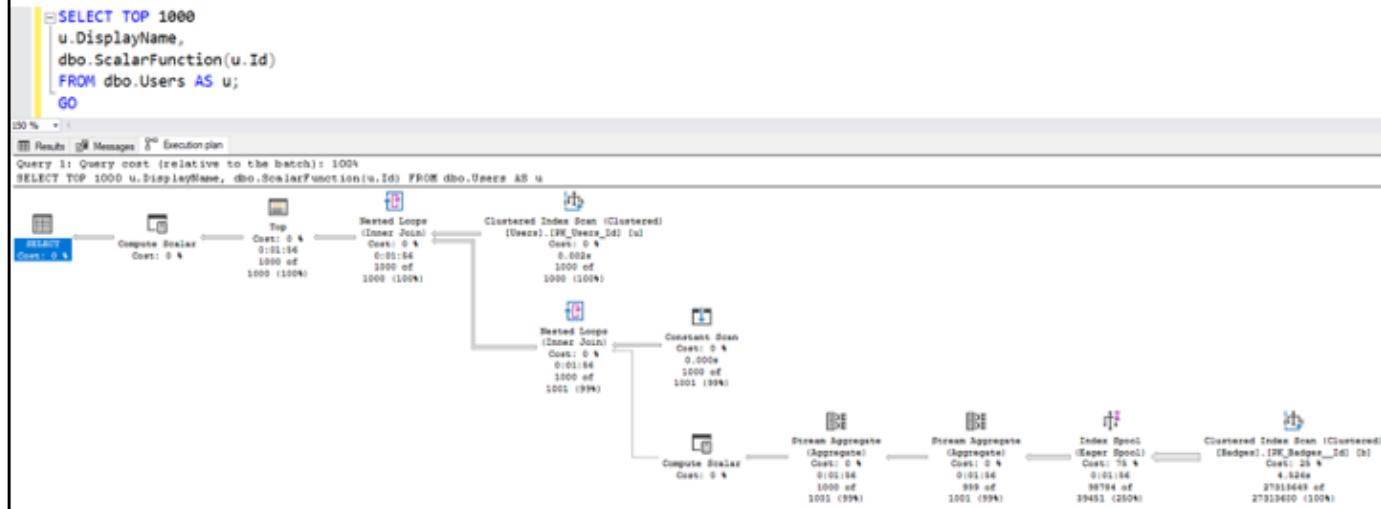
But it's not good: that's a hell of a lot of reads.

2.2 p49

## The plan shows what's up, too.

The Badges table now shows up, but...

We get an Index Spool (Eager Spool) and no parallelism.



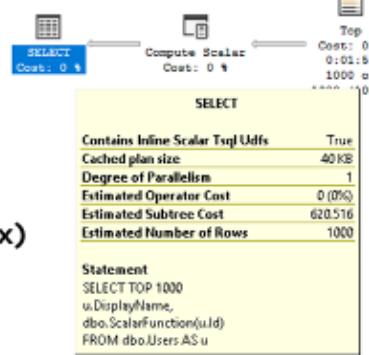
## What went wrong

SQL Server 2019 inlined the function, but:

- Chose an index spool plan (that didn't suggest an index)
- Only used a single CPU core (whereas 2017 let the function itself go parallel)
- And it wasn't due to low query cost either

If inlining doesn't work out for your function (some go slower), you can use WITH INLINE = OFF at the function level:

```
CREATE OR ALTER FUNCTION dbo.discount_price(@price DECIMAL(12,2), @discount DECIMAL(12,2))
RETURNS DECIMAL (12,2)
WITH INLINE = OFF
AS
BEGIN
    RETURN @price * (1 - @discount);
END
```



2.2 p51

**And you can still do way  
better yourself.**



2.2 p52



# Rewriting your UDFs

Switching to single-statement inline ones or APPLY

## If you're facing MSTVFs that look like this:

```
-- Multi Statement Table Valued Function
CREATE FUNCTION dbo.MultiStatementTVF ( @uid INT )
RETURNS @Out TABLE ( BadgeCount BIGINT )
    WITH SCHEMABINDING
AS
BEGIN
    INSERT INTO @Out(BadgeCount)
    SELECT COUNT_BIG(*) AS BadgeCount
    FROM dbo.Badges AS b
    WHERE b.UserId = @uid
    GROUP BY b.UserId;
    RETURN;
END;
```



2.2 p54

**And you call them like this:**

```
SELECT TOP 1000
    u.DisplayName,
    mi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.MultiStatementTVF(u.Id) AS mi
```



2.2 p55

Then they'll fly if you make 'em look like this:

```
-- Inline Table Valued Function
CREATE FUNCTION dbo.InlineTVF ( @uid INT )
RETURNS TABLE
    WITH SCHEMABINDING
AS
RETURN
    SELECT COUNT_BIG(*) AS BadgeCount
    FROM dbo.Badges AS b
    WHERE b.UserId = @uid
    GROUP BY b.UserId;
```



2.2 p56

## Single-statement inline TVFs

If you can write it with a single SELECT, then SQL Server can inline it just like it was a view.

The good: you only have to change the function's contents, but not the way you call the function. The callers stay the same.

The bad: complex MSTVFs can turn into a hell of CASEs.

The ugly: if you can't get it into one statement, it won't perform. Your only options are things like stored procedures.



2.2 p57

## lol.exe is terminating unexpectedly

```
SELECT TOP 1000
    u.DisplayName,
    bi.BadgeCount
    FROM dbo.Users AS u
    CROSS APPLY dbo.InlineTVF(u.Id) AS bi
```



2.2 p58

```
SELECT TOP 1000
    u.DisplayName,
    bi.BadgeCount
FROM dbo.Users AS u
CROSS APPLY dbo.InlineTVF(u.Id) AS bi;
GO
```

**SEVEN SECONDS**

150 % < >

Results Messages Execution plan

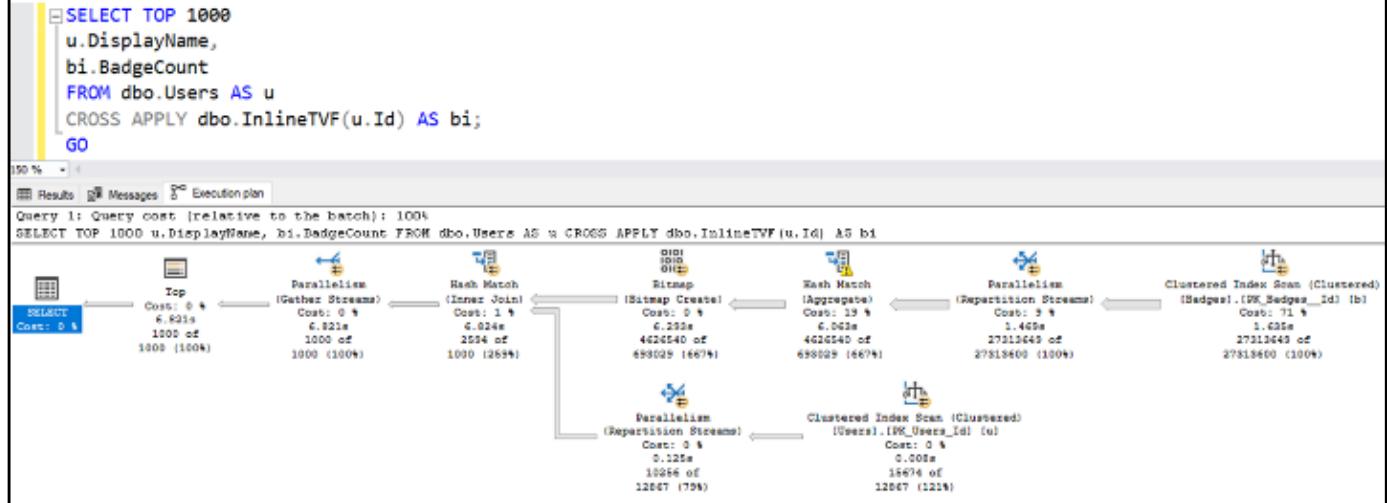
	DisplayName	BadgeCount
1	Community	1
2	Tom	104
3	Karl Seguin	86
4	Justin	100
5	Daren Kopp	159
6	svrist	96
7	Kalid	93

Query executed... | SQL2019 (15.0 RTM) | SQL2019\Administrator ... | StackOverflow | 00:00:07 | 1,000 rows    2.2 p59

# The plan gets way better

We get parallelism, and no index spool.

And if you tuned indexes, you could do even better.



## **So many good benefits here**

**Parallelism**

**One call to the function**

**Done in seconds, not minutes**

**Accurate STATISTICS IO**

**Accurate query cost**



**2.2 p61**

## CROSS APPLY would work too

```
SELECT TOP 1000
    u.DisplayName ,
    bca.BadgeCount
FROM      dbo.Users AS u
CROSS APPLY ( SELECT b.UserId ,
                    COUNT_BIG(*) AS BadgeCount
                FROM      dbo.Badges AS b
                WHERE     u.Id = b.UserId
                GROUP BY b.UserId
            ) bca;
```



2.2 p62

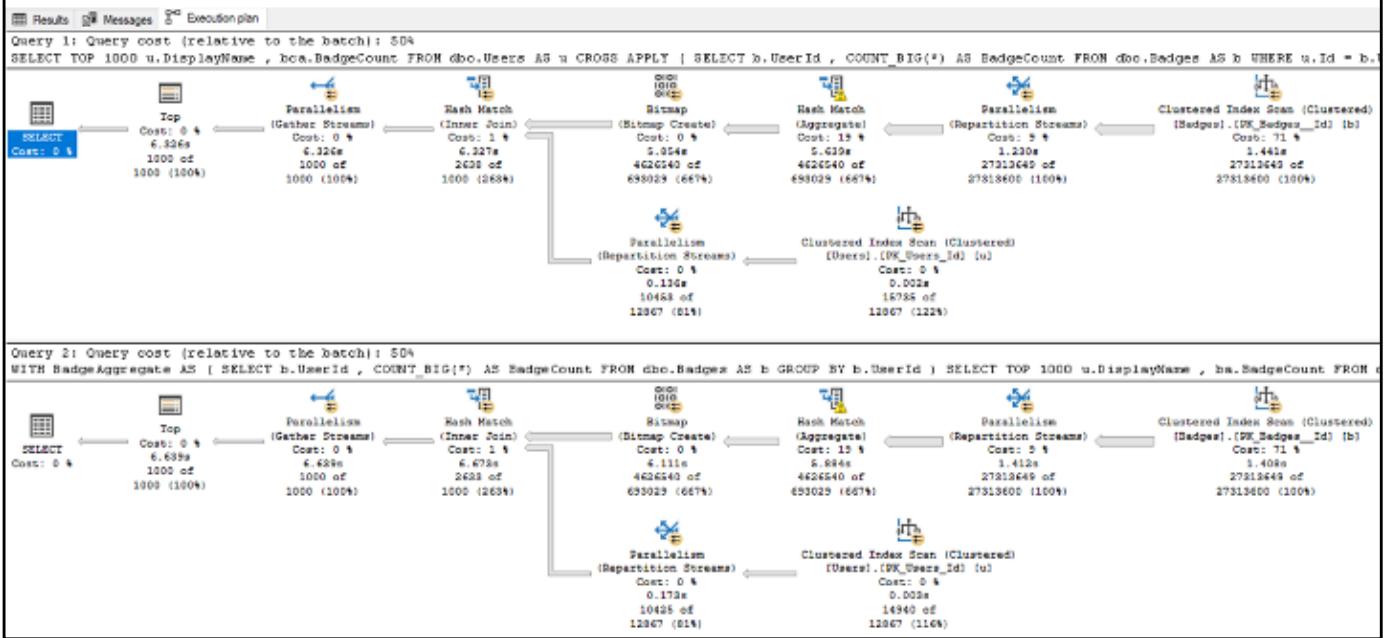
## As would a CTE

```
WITH BadgeAggregate AS
( SELECT b.UserId ,
   COUNT_BIG(*) AS BadgeCount
  FROM    dbo.Badges AS b
 GROUP BY b.UserId )
SELECT TOP 1000
       u.DisplayName ,
       ba.BadgeCount
  FROM    dbo.Users AS u
 JOIN    BadgeAggregate AS ba
  ON      ba.UserId = u.Id;
```



2.2 p63

## Same plan, same costs, etc.



## But if you're facing scalar functions...

```
-- Scalar Function
CREATE FUNCTION dbo.ScalarFunction ( @uid INT )
RETURNS BIGINT
    WITH RETURNS NULL ON NULL INPUT,
         SCHEMABINDING
AS
BEGIN
    DECLARE @BCount BIGINT;
    SELECT @BCount = COUNT_BIG(*)
    FROM dbo.Badges AS b
    WHERE b.UserId = @uid
    GROUP BY b.UserId;
    RETURN @BCount;
END;
```



2.2 p65

## **SQL Server 2017 & prior can't inline these.**

**Even if they only have one statement, they won't go inline.**

**That means you HAVE to get rid of them altogether, not just tune them.**

**Which is problematic since you have to touch every query that has 'em.**

**Thus the excitement about SQL Server 2019.**



**2.2 p66**



**Finding the little buggers**

## Find queries executed frequently

```
EXEC sp_BlitzCache @SortOrder = 'xpm'  
GO  
EXEC sp_BlitzCache @SortOrder = 'executions'  
  
/* Sorts your plan cache by highest executions per  
minute or executions.  
  
When this is high, it's sometimes a sign of row-by-  
row functions being called, as we saw.  
  
http://www.brentozar.com/blitzcache/ */
```



2.2 p68

## Or you can rip through code...

```
/*You can look for bad functions like this*/  
  
SELECT SCHEMA_NAME(o.schema_id) AS [schema_name], o.name, o.type_desc  
FROM sys.objects AS o  
WHERE o.type IN ('FN', N'TF')  
AND o.is_ms_shipped = 0;  
  
/*Plug the names in here to search stored proc text*/  
  
SELECT obj.name ,  
       sc.text  
  FROM sys.objects obj  
INNER JOIN sys.syscomments sc  
  ON sc.id = obj.object_id  
 WHERE obj.type = 'P'  
   AND sc.text LIKE '% FUNCTION NAME %'
```



2.2 p69

## Recap

**Functions sound like a great idea at first**

- Code reusability is a best practice everywhere else

**But right now, functions cause problems in SQL Server**

- Serializing execution
- Row by row execution

**Inline table valued functions reduce these issues**

- Treated like a view or CTE rather than a separate procedural task



2.2 p70