



**BRENT OZAR**  
UNLIMITED®

# The New Robots in SQL Server 2017 & 2019

3.4 p1



I keep hearing  
about these robots,  
and about how they're gonna take my job.



3.4 p2



I used SQL 2019  
this whole class.

I rarely hear any of you say,  
“The robots fixed it all for me.”



3.4 p3

**2017 & 2019's robots are kinda  
like your phone's AutoCorrect.**

You're probably not going to  
disable them, but...

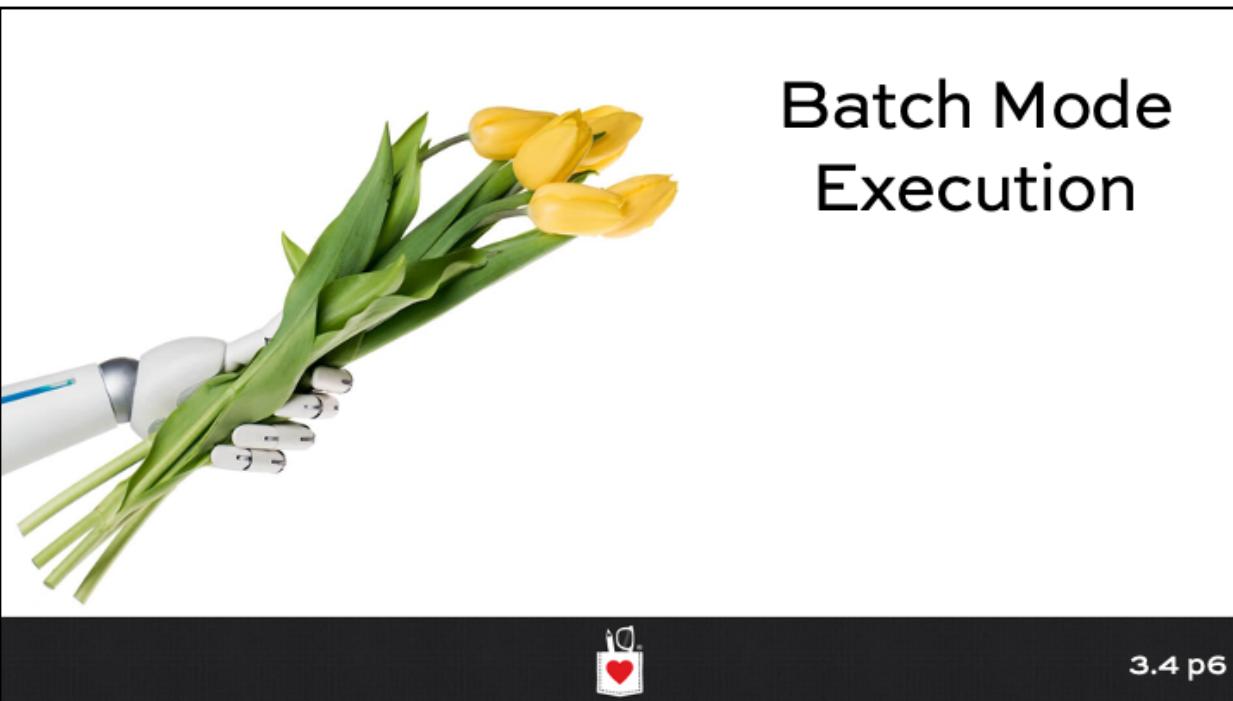


3.4 p4

	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Automatic tuning</b>										
Query Store automatic plan correction	Yes	Yes	Enterprise	140	Yes	-	-	Yes	-	OK
Automatic index management	-	-	-	140	Yes	-	-	Yes	-	OK
<b>Batch mode execution</b>										
For columnstore indexes	Yes	Yes	Standard	140	Yes	-	Kinda	Yes	-	Great!
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	-	Great!
<b>Functions</b>										
TVF interleaved execution	Yes	Yes	Standard	140	Yes	-	Yes	Yes	-	OK
Scalar function inlining	-	Yes	Standard	150	-	-	Yes	Yes	Yes	Test it hard
<b>Memory grant feedback (adaptive grants)</b>										
For columnstore indexes	-	Yes	Enterprise	140	Yes	-	Kinda	Yes	-	OK
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	Yes	Test it hard
<b>Adaptive joins</b>										
For plans with columnstore indexes	Yes	Yes	Enterprise	140	Yes	-	Yes	Yes	-	OK
For all plans	-	Yes	Enterprise	150	-	-	Yes	Yes	-	OK
<b>Others</b>										
Approximate Count Distinct	-	Yes	Standard	150	Yes	Yes	Yes	Yes	-	OK
Table variable deferred compilation	-	Yes	Standard	150	Yes	-	Yes	Yes	-	OK



3.4 p5



## Reporting-style query on 2017:

```
72 SET STATISTICS IO, TIME ON;
73 ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 140;
74 GO
75 SELECT YEAR(v.CreationDate) AS CreationYear, MONTH(v.CreationDate) AS CreationMonth,
76     COUNT(*) AS VotesCount,
77     AVG(BountyAmount * 1.0) AS AvgBounty
78 FROM dbo.Votes v
79 GROUP BY YEAR(v.CreationDate), MONTH(v.CreationDate)
80 ORDER BY YEAR(v.CreationDate), MONTH(v.CreationDate)
81 GO
```

Results Messages Execution plan

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

```
SELECT [CreationYear] AS CreationYear, [CreationMonth] AS CreationMonth, COUNT(*) AS VotesCount, AVG(BountyAmount * 1.0) AS AvgBounty FROM dbo.Votes v GROUP BY YEAR(v.CreationDate), MONTH(v.CreationDate)
```

The execution plan diagram illustrates the flow of data through various operators. It starts with a 'SELECT' operator, followed by a 'Compute Scalar' (Gather Streams) operator with a cost of 0.4%. This is followed by a 'Stream Aggregate' operator with a cost of 0.4%, which then feeds into a 'Sort' operator with a cost of 0.4%. The 'Sort' operator leads to a 'Partition Scan (Separation Streams)' operator with a cost of 0.4%. This is followed by a 'Partial Aggregate' operator with a cost of 44.4%, which then feeds into a 'Hash Match (Partial Aggregate)' operator with a cost of 4.104%. Finally, the process ends with a 'Compute Scalar' operator with a cost of 1.4% and a 'Clustered Index Scan (Clustered) [Votes]' operator with a cost of 83.4%.

3.4 p7

## Lots of reads, lots of CPU work.

```
72 SET STATISTICS IO, TIME ON;
73 ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 140;
74 GO
75 SELECT YEAR(v.CreationDate) AS CreationYear, MONTH(v.CreationDate) AS CreationMonth,
76     COUNT(*) AS VotesCount,
77     AVG(BountyAmount * 1.0) AS AvgBounty
78     FROM dbo.Votes v
79     GROUP BY YEAR(v.CreationDate), MONTH(v.CreationDate)
80     ORDER BY YEAR(v.CreationDate), MONTH(v.CreationDate)
81 GO
```

150 % ▾

Results Messages Execution plan

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead
Table 'Votes'. Scan count 5, logical reads 693459, physical reads 0, page server reads 0, read-ahead
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row affected)
```

```
SQL Server Execution Times:
    CPU time = 60969 ms, elapsed time = 15794 ms
```

## Change the compat level to 2019:

```
84 /* Then try in 2019: */
85 ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 150;
86 GO
87 SELECT YEAR(v.CreationDate) AS CreationYear, MONTH(v.CreationDate) AS CreationMonth,
88 COUNT(*) AS VotesCount,
89 AVG(BountyAmount * 1.0) AS AvgBounty
90 FROM dbo.Votes v
91 GROUP BY YEAR(v.CreationDate), MONTH(v.CreationDate)
92 ORDER BY YEAR(v.CreationDate), MONTH(v.CreationDate)
93 GO
```

Execution plan details:

Operation	Cost	Rows	Approximate Run Time
SELECT	0	132	6.741s
Gather Streams	0	132	0.001s
Sort	0	132	0.000s
Compute Scalar	0	132	0.000s
Hash Match (Aggregate)	8	132	2.435ms
Compute Scalar	0	132	3.612ms
Clustered Index Scan [Votes] [PK_Votes_Id] [v]	82	150784000	2.161s

## Same reads, but less CPU & time!

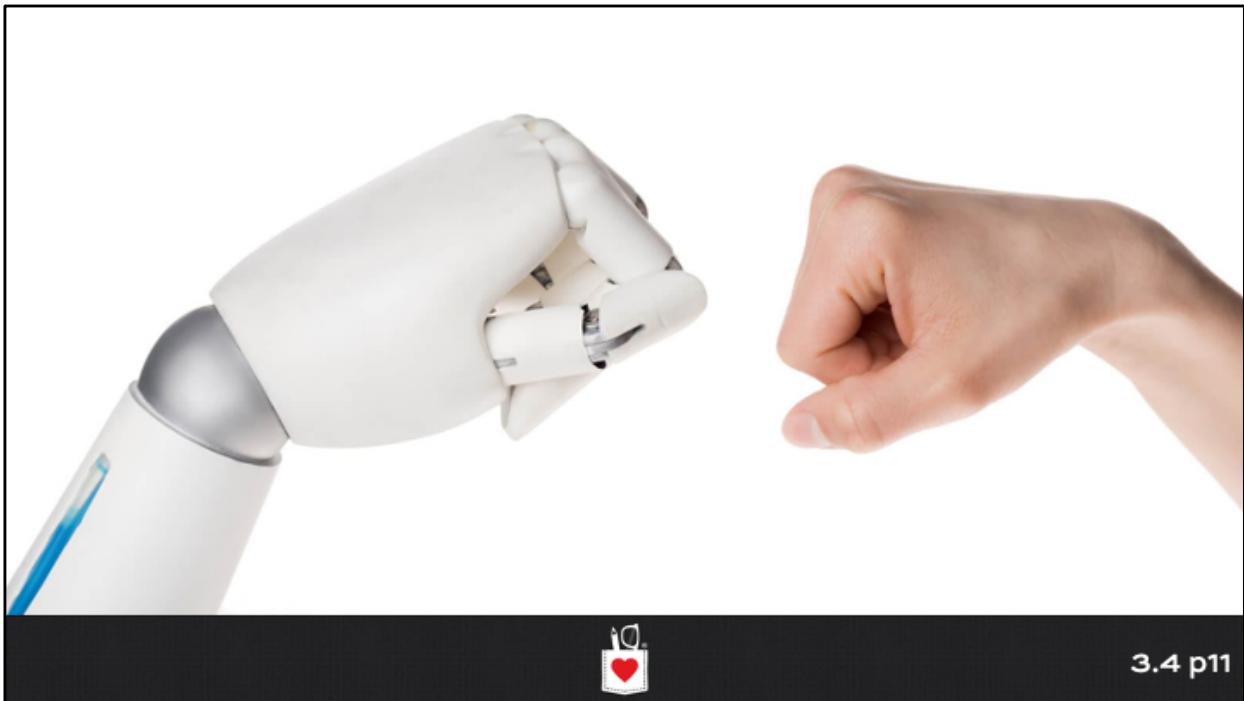
2017 was:

```
Table 'Workfile'. Scan count 0, logical reads 0.  
Table 'Votes'. Scan count 5, logical reads 693459,  
Warning: Null value is eliminated by an aggregate or  
(1 row affected)  
SQL Server Execution Times:  
CPU time = 60969 ms, elapsed time = 15794 ms.
```

2019:

```
Table 'Votes'. Scan count 5, logical reads 693884,  
Table 'Worktable'. Scan count 0, logical reads 0, p  
Warning: Null value is eliminated by an aggregate o  
(1 row affected)  
SQL Server Execution Times:  
CPU time = 22313 ms, elapsed time = 6823 ms.
```

3.4 p10



3.4 p11

# It's not just a simpler plan:

2017 plan has two parallelism zones:



2019 plan – only one parallelism zone:



3.4 p12

# It's different operators, really:

2017's table scan:

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	150784380
Actual Number of Rows for All Executions	150784380
Actual Number of Batches	0

2019's:

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Number of Rows Read	150784380
Actual Number of Rows for All Executions	150784380
Actual Number of Batches	167541

3.4 p13

## **2017 & prior: row mode execution**

Each plan operator is a mini-program

It runs in isolation

It passes data to the next operator in the form of **rows**



3.4 p14

## Then came columnstore indexes.

The data is stored differently.

I describe this in Fundamentals of Columnstore.

Paper: Query Execution in Column-Oriented Database Systems by Daniel J. Abadi:

<http://www.cs.umd.edu/~abadi/papers/abadiphd.pdf>

Niko Neugebauer 100+part blog series:

<http://www.nikoport.com/columnstore/>



3.4 p15

## Why this mattered

Think of it as a separate index on each column:

```
□ dbo.Votes
  □ Columns
    □ Id (PK, int, not null)
    □ PostId (int, not null)
    □ UserId (int, null)
    □ BountyAmount (int, null)
    □ VoteTypeId (int, not null)
    □ CreationDate (datetime, not null)
```

But here's the trick question:  
when do you reassemble these columns  
into their original rows?



3.4 p16

## Batch mode execution means

This operator isn't rebuilding the rows yet:  
it's passing groups of columns to the next operator.

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Number of Rows Read	150784380
Actual Number of Rows for All Executions	150784380
Actual Number of Batches	167541



3.4 p17

## Like an index on every column

Think of it as a separate index on each column:

```
□ dbo.Votes
  □ Columns
    □ Id (PK, int, not null)
    □ PostId (int, not null)
    □ UserId (int, null)
    □ BountyAmount (int, null)
    □ VoteTypeId (int, not null)
    □ CreationDate (datetime, not null)
```

But here's the trick question:  
when do you reassemble these columns  
into their original rows?



3.4 p18

## So many cool implications

Batches of columnar data can be:

- Way smaller due to duplication  
(only unique values are stored)
- Way easier to group/sum/sort on  
(since so much is duplicated)

Result: massively less CPU work  
for reporting queries (table scans  
with grouping, summing, formulas.)

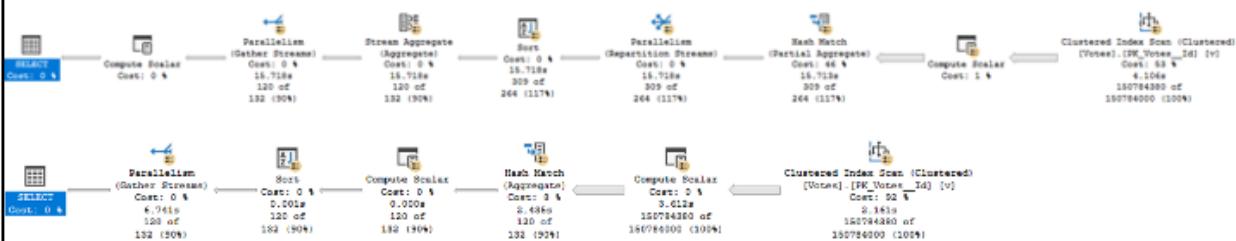
dbo.Votes
Columns
Id (PK, int, not null)
PostId (int, not null)
UserId (int, null)
BountyAmount (int, nul)
VoteTypeId (int, not nul)
CreationDate (datetime,



3.4 p19

# The first challenge

Which of these plans is getting batch mode?



And which of the operators are getting it?

3.4 p20



## More challenges

If you got batch mode on an operator:

- Is it working out in your favor?
- Why was it chosen?
- How can you change it?

If you didn't get batch mode:

- Why not?
- Can you change things to get it?



3.4 p21



Still, I like it.  
A lot.



	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Batch mode execution</b>										
For columnstore indexes	Yes	Yes	Standard	140	Yes	-	Kinda	Yes	-	Great!
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	-	Great!

Columnstore gets it in Standard, but it just works on columnstore indexes, period. No big gotchas there.

Rowstore tables only get it in Enterprise Edition.

I think that's a perfect fit for batch mode:

- It makes the most sense in reports
- It makes the least sense in OLTP



3.4 p23

# TVF Interleaved Execution



3.4 p24

## Multi-Statement Table-Valued Functions

We covered why these suck:

- Hard to see impact in the query plan, stats IO
- Can run row-by-row
- Estimates are hard-coded garbage (100 or 1)

2017 can fix the latter by running the MSTVF first, then passing the row estimates to the rest of the plan.

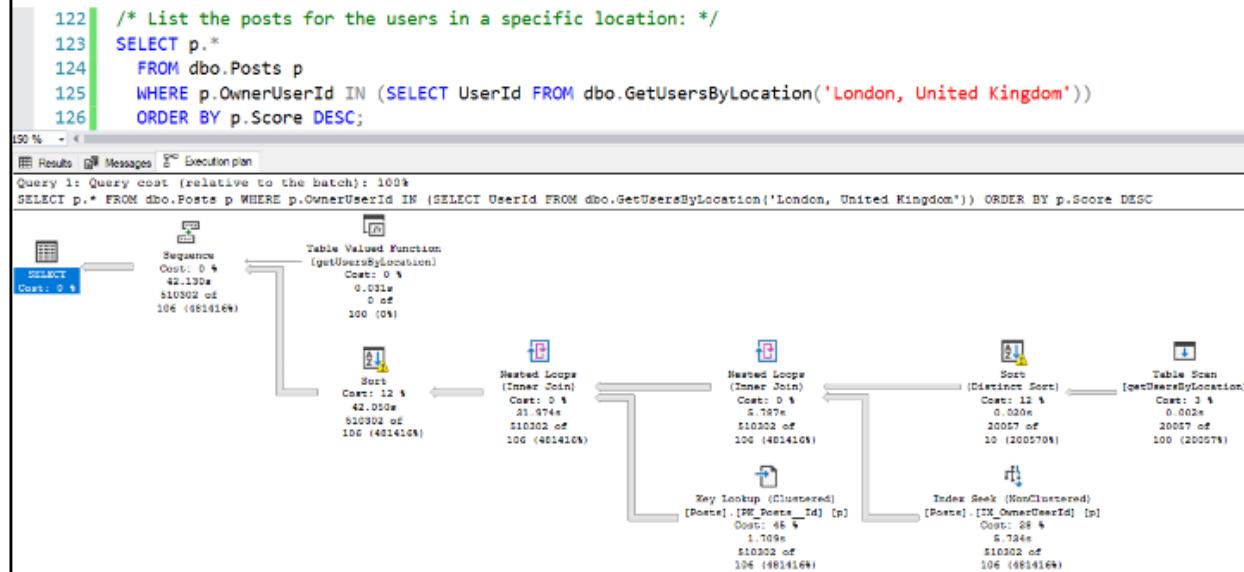


3.4 p25

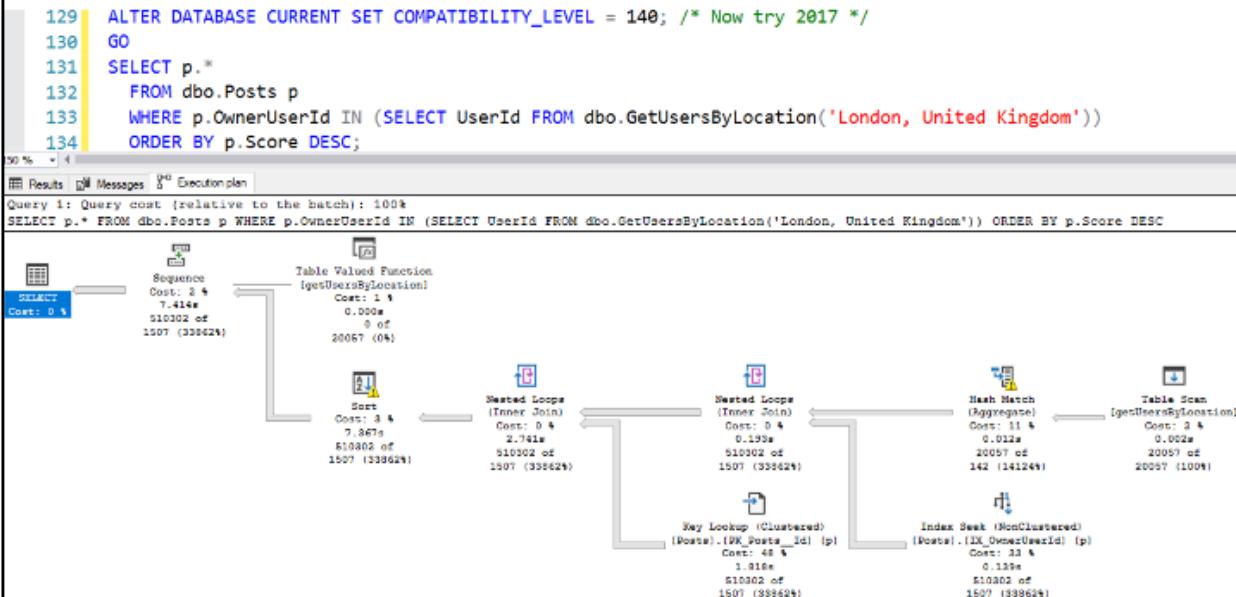
```
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 130; /* 2016 at first */
GO
CREATE OR ALTER FUNCTION dbo.getUsersByLocation ( @Location NVARCHAR(100) )
RETURNS @Out TABLE ( UserId INT )
WITH SCHEMABINDING
AS
BEGIN
    INSERT INTO @Out(UserId)
    SELECT Id
    FROM dbo.Users
    WHERE Location = @Location;
    RETURN;
END;
GO

/* List the posts for the users in a specific location: */
SELECT p.*
FROM dbo.Posts p
WHERE p.OwnerUserId IN (SELECT UserId FROM dbo.GetUsersByLocation('London, United Kingdom'))
ORDER BY p.Score DESC;
```

## It takes 42 seconds

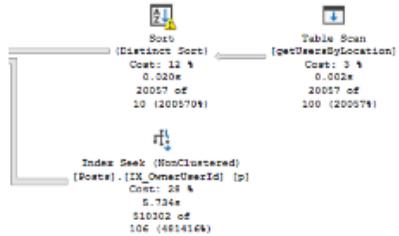


## 2017 compat mode: 7 seconds!

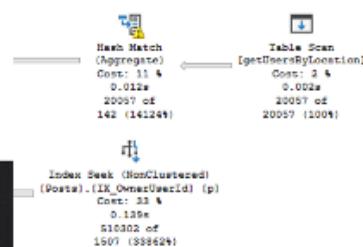


# The difference

2016 estimates a hard-coded 100 rows from the TVF:



But 2017+ executes it,  
and knows 20,057  
rows will come out:



3.4 p29



## Interleaved Execution is like phone-a-friend.

It gives a heads up to the rest of the plan that a whole lot of rows are incoming, thereby improving memory grant accuracy. That's great!



3.4 p30



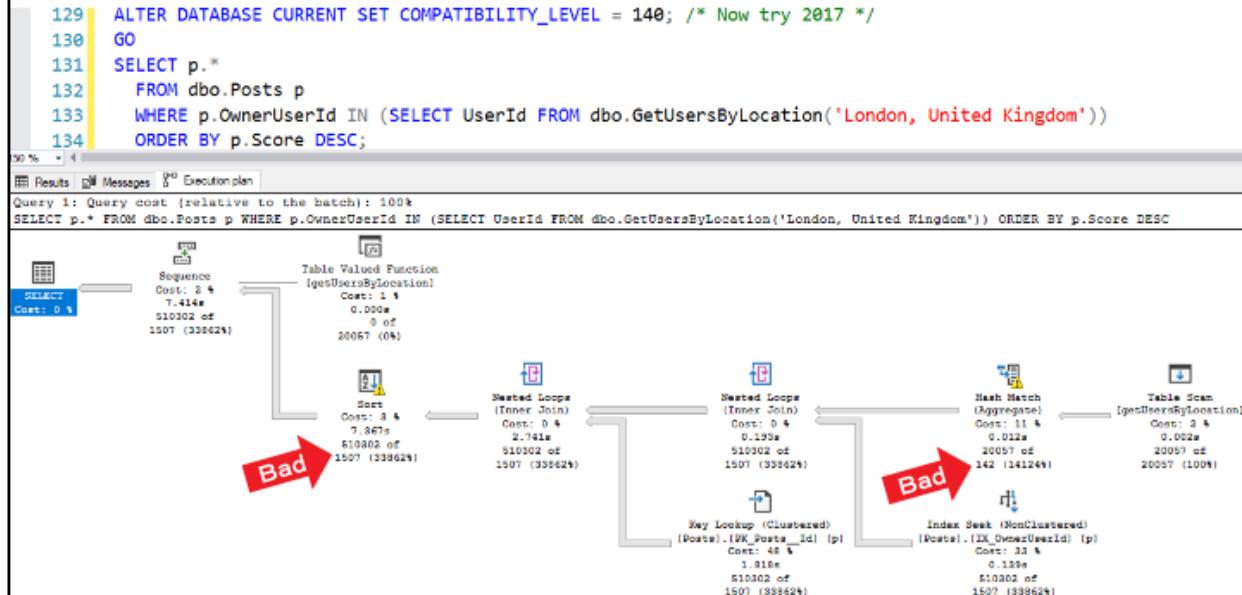
## You still shouldn't be using MSTVF<sub>s</sub> though.

While we know the *number* of rows coming out, we still don't have great stats on their contents.



3.4 p31

## The sort & hash match both spill.



	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Functions</b>										
TVF interleaved execution	Yes	Yes	Standard	140	Yes	-	Yes	Yes	-	OK
Scalar function inlining	-	Yes	Standard	150	-	-	Yes	Yes	Yes	Test it hard

TVF interleaved execution is fine.

It makes a bad thing (TVFs) suck less.

Scalar function inlining, that's different.



3.4 p33

# Scalar Function Inlining

One of the most ambitious things I've  
ever seen in a database server.



3.4 p34

## The idea

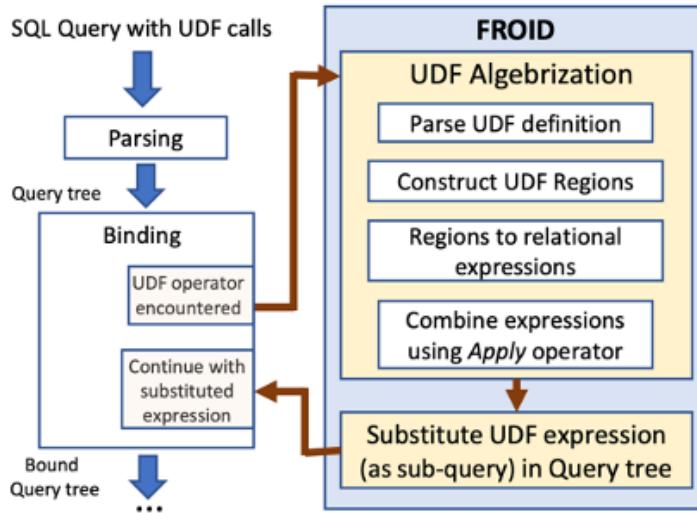
Take scalar user-defined functions that have always been designed to run on a row-by-row basis

1. Rewrite the UDFs automatically into set-based functions (think: turn them into a view you can join to)
2. Rewrite queries calling the UDFs so that they can leverage the set-based version

White paper: <https://arxiv.org/abs/1712.00498>  
(Click the PDF link in the Download box)



3.4 p35



**Figure 3:** Overview of the Froid framework



3.4 p36

## This paper is fun (for me) to read

Imperative Statement (T-SQL)	Relational expression (T-SQL)
DECLARE {@var data_type [= expr]}[,... n];	SELECT {expr null AS var}{,... n};
SET {@var = expr}{,... n};	SELECT {expr AS var}{,... n};
SELECT {@var1 = prj.expr1}{,... n} FROM sql_expr;	{SELECT prj.expr1 AS var1 FROM sql_expr}; [,... n]
IF (pred_expr) {t_stmt; [...] n} ELSE {f_stmt; [...] n}}	SELECT CASE WHEN pred_expr THEN 1 ELSE 0 END AS pred_val; {SELECT CASE WHEN pred_val = 1 THEN t_stmt ELSE f_stmt; }[... n]
RETURN expr;	SELECT expr AS returnVal;

Table 1: Relational algebraic expressions for imperative statements (using standard T-SQL notation from [34])



3.4 p37

## I was amazed that this shipped.

It's the most ambitious thing I've seen in SQL Server.

Other features were harder and more complex:  
CLR, Hekaton, Polybase, Big Data Clusters.

But those had a limited blast radius because they  
required you to change your code, tables, indexes.

This feature was supposed to work  
without you changing anything at all.



3.4 p38

## Well, that was the theory.

But they keep finding bugs with it, and disabling it:

### 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).

Microsoft SQL Server 2019 introduces the Scalar UDF Inlining feature that can improve the performance of queries that invoke T-SQL Scalar UDFs, where UDF execution is the main bottleneck. T-SQL Scalar UDF Inlining automatically transforms inlineable UDFs into relational expressions.



3.4 p39

## At first, they fixed bugs...

This cumulative update includes several **fixes** across the following areas for scenarios in which a query that uses Scalar UDF Inlining may return an error or unexpected results:

- Type mismatch error if the return type of the UDF is a sql\_variant (added in Microsoft SQL Server 2019 CU2)
- UDF invocation from sp\_executesql aborts execution (added in Microsoft SQL Server 2019 CU2)
- UDFs referencing labels without an associated GOTO command return incorrect results (added in Microsoft SQL Server 2019 CU2)
- Out-of-memory conditions and memory leaks occur because of very large scalar UDFs (added in Microsoft SQL Server 2019 CU2)
- Uninitialized variables used in condition (IF-ELSE) statements cause errors (added in Microsoft SQL Server 2019 CU2)

CU2 had bug fixes, but after that, they just started disabling scalar function inlining in more scenarios...



3.4 p40

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)

## And even when it “works”

It doesn't always speed up queries, as we saw earlier.

Thus my verdict:

	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
Functions										
TVF interleaved execution	Yes	Yes	Standard	140	Yes	-	Yes	Yes	-	OK
Scalar function inlining	-	Yes	Standard	150	-	-	Yes	Yes	Yes	Test it hard

It might work for you, but...  
I've seen it backfire, HARD.



3.4 p42

Taryn aka bluefeet @tarynpivots · Feb 14  
I guess it's good I didn't finish all our SQL 2019 upgrades. This includes a fix on a bug we have been hitting with UDF inlining.

Aaron Bertrand @AaronBertrand · Feb 13  
SQL Server 2019 Cumulative Update #2 is available. The build is 15.0.4013.40 and there are 134 total fixes and enhancements.  
[sqlblog.org/2020/02/14/sql...](http://sqlblog.org/2020/02/14/sql...)

Taryn aka bluefeet @tarynpivots · Feb 14  
I'm sad to report our issue is still not fixed with this CU. Our use of UDF inlining is still broken and will stay disabled across all of our databases.

## Taryn is Stack Overflow's DBA.

It's been rough seeing them struggle with 2019 performance issues.

3.4 p43

# Adaptive Memory Grants



3.4 p44

## Query workspace memory grants

It's really hard for SQL Server to get estimates right.

It's not SQL Server's fault:  
it's usually our crappy queries and indexes.

SQL Server does track how often spills occur for a given query plan, but it just doesn't do anything with that data.



<https://docs.microsoft.com/en-us/sql/relational-databases/performance/adaptive-query-processing>

#### Memory grant feedback sizing

For an excessive memory grant condition, if the granted memory is more than two times the size of the actual used memory, memory grant feedback will recalculate the memory grant and update the cached plan. Plans with memory grants under 1 MB will not be recalculated for overages. For an insufficiently sized memory grant condition, that result in a spill to disk for batch mode operators, memory grant feedback will trigger a recalculation of the memory grant. Spill events are reported to memory grant feedback and can be surfaced via the *spilling\_report\_to\_memory\_grant\_feedback* xEvent. This event returns the node id from the plan and spilled data size of that node.

#### Memory grant feedback and parameter sensitive scenarios

Different parameter values may also require different query plans in order to remain optimal. This type of query is defined as "parameter-sensitive." For parameter-sensitive plans, memory grant feedback will disable itself on a query if it has unstable memory requirements. The plan is disabled after several repeated runs of the query and this can be observed by monitoring the *memory\_grant\_feedback\_loop\_disabled* xEvent. For more information about parameter sniffing and parameter sensitivity, refer to the [Query Processing Architecture Guide](#).

3.4 p46

## Memory grant feedback caching

Feedback can be stored in the cached plan for a single execution. It is the consecutive executions of that statement, however, that benefit from the memory grant feedback adjustments. This feature applies to repeated execution of statements. Memory grant feedback will change only the cached plan. Changes are currently not captured in the Query Store. Feedback is not persisted if the plan is evicted from cache. Feedback will also be lost if there is a failover. A statement using `OPTION (RECOMPILE)` creates a new plan and does not cache it. Since it is not cached, no memory grant feedback is produced and it is not stored for that compilation and execution. However, if an equivalent statement (that is, with the same query hash) that did **not** use `OPTION (RECOMPILE)` was cached and then re-executed, the consecutive statement can benefit from memory grant feedback.

## Tracking memory grant feedback activity

You can track memory grant feedback events using the `memory_grant_updated_by_feedback` xEvent. This event tracks the current execution count history, the number of times the plan has been updated by memory grant feedback, the ideal additional memory grant before modification and the ideal additional memory grant after memory grant feedback has modified the cached plan.



3.4 p47

## Seeing it in action

Build our stored proc that is vulnerable to sniffing:

```
/* Adaptive memory grants for rowstore tables */
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 140; /* 2017 */
GO
CREATE OR ALTER PROC dbo.usp_UsersByReputation @Reputation INT AS
    SELECT TOP 100000 u.*
        FROM dbo.Users u
        WHERE u.Reputation = @Reputation
        ORDER BY u.DisplayName;
GO
```

Then run it back & forth...



3.4 p48

```
/* Adaptive memory grants for rowstore tables */
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 140; /* 2017 */
GO
CREATE OR ALTER PROC dbo.usp_UsersByReputation @Reputation INT AS
    SELECT TOP 100000 u.*
        FROM dbo.Users u
        WHERE u.Reputation = @Reputation
        ORDER BY u.DisplayName;
GO

/* Free the plan cache and run it for tiny data */
DBCC FREEPROCCACHE;
GO
EXEC usp_UsersByReputation @Reputation = 2; /* Warns about granting too much RAM */
GO
EXEC usp_UsersByReputation @Reputation = 1; /* Spills to disk */
GO
/* Now do it in the opposite order: */
DBCC FREEPROCCACHE;
GO
EXEC usp_UsersByReputation @Reputation = 1; /* Grants a ton of memory */
GO
EXEC usp_UsersByReputation @Reputation = 2; /* Leaves a ton of unused memory on the floor */
GO
```

p49

```

151 /* Free the plan cache and run it for tiny data */
152 DBCC FREEPROCCACHE;
153 GO
154 EXEC usp_UsersByReputation @Reputation = 2; /* Warns about granting too much RAM */
155 GO
156 EXEC usp_UsersByReputation @Reputation = 1; /* Spills to disk */
157 GO

```

50 % + 4

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 50%

SELECT TOP 100000 u.\* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY u.DisplayName

Missing Index (Impact 97.5955): CREATE NONCLUSTERED INDEX [Name of Missing Index, sysname,>] ON [dbo].[Users] ([Reputation])

Sort (Top 100000) Cost: 2 %  
Cost: 0 %  
9149 of 9149 (100%)

Nested Loops (Index Scan) Cost: 0 %  
Cost: 0 %  
9149 of 9149 (100%)

Index Seek (NonClustered) ([Users].[IX\_Reputation] [u]) Cost: 0 %  
Cost: 0 %  
9149 of 9149 (100%)

Key Lookup (Clustered) ([Users].[PK\_Users\_Id] [u]) Cost: 99 %  
Cost: 0 %  
9149 of 9149 (100%)

Query 2: Query cost (relative to the batch): 50%

SELECT TOP 100000 u.\* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY u.DisplayName

Missing Index (Impact 97.5955): CREATE NONCLUSTERED INDEX [Name of Missing Index, sysname,>] ON [dbo].[Users] ([Reputation])

SELECT Cost: 0 %  
Cost: 0 %  
9149 (100%)

Sort (Top 100000) Cost: 2 %  
Cost: 2 %  
22.530e 100000 of 9149 (44047%)

Nested Loops (Index Scan) Cost: 0 %  
Cost: 0 %  
9.167e 6044557 of 9149 (44047%)

Index Seek (NonClustered) ([Users].[IX\_Reputation] [u]) Cost: 0 %  
Cost: 0 %  
9.710e 6044557 of 9149 (44047%)

Key Lookup (Clustered) ([Users].[PK\_Users\_Id] [u]) Cost: 99 %  
Cost: 0 %  
6044557 of 9149 (44047%)

3.4 p50

```

151 /* Free the plan cache and run it for tiny data */
152 DBCC FREEPROCCACHE;
153 GO
154 EXEC usp_UsersByReputation @Reputation = 2; /* Warning */
155 GO
156 EXEC usp_UsersByReputation @Reputation = 1; /* Spill */
157 GO

```

Results pane showing the execution plan for Query 1. The plan shows a Sort operator (Top N Sort) with a cost of 2.4. The logical operation is Actual Execution Mode. Estimated execution mode shows 100000 rows. Actual number of rows for all executions is 9149. Estimated CPU cost is 0.485177. Estimated Subtree Cost is 29.6857. Number of executions is 1. Estimated number of executions is 1. Estimated number of rows per execution is 9149. Estimated row size is 4468.8.

Query 2: SELECT TOP 100000 u.\* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY Missing Index (Impact 97.5)

Results pane showing the execution plan for Query 2. The plan shows a Sort operator (Top N Sort) with a cost of 2.4. The logical operation is Actual Execution Mode. Estimated execution mode shows 100000 rows. Actual number of rows for all executions is 9149. Estimated CPU cost is 0.485177. Estimated Subtree Cost is 29.6857. Number of executions is 1. Estimated number of executions is 1. Estimated number of rows per execution is 9149. Estimated row size is 4468.8.

Actual Rebinds: 1. Actual Rewinds: 0. Ion ORDER BY Missing Ind Node ID: 0.

Output List: [StackOverflow].[dbo].[User].Id, [StackOverflow].[dbo].[User].AboutMe, [StackOverflow].[dbo].[User].Age, [StackOverflow].[dbo].[User].CreationDate, [StackOverflow].[dbo].[User].DisplayName, [StackOverflow].[dbo].[User].EmailHash, [StackOverflow].[dbo].[User].EmailVerified, [StackOverflow].[dbo].[User].Location, [StackOverflow].[dbo].[User].Reputation, [StackOverflow].[dbo].[User].Views, [StackOverflow]...

Warnings: Operator used tempdb to spill data during execution with spill level 1 and 1 spilled thread(s). Sort wrote 83481 pages to and read 83481 pages from tempdb with granted memory 50856KB and used memory 50856KB.

Order By: [StackOverflow].[dbo].[User].DisplayName Ascending.

Query executed successfully.

## That's a big spill.

2017 compat level:  
83,481 pages spilled to TempDB.  
(And the query's slow, duh.)

Let's try 2019 compat level.

3.4 p51

```

167 ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 150; /* 2019 */
168 GO
169 DBCC FREEPROCCACHE;
170 GO
171 EXEC usp_UsersByReputation @Reputation = 2;
172 GO
173 EXEC usp_UsersByReputation @Reputation = 1;
174 GO

```

Results tab shows the execution plan for Query 1.

Query 1: Query cost (relative to the batch): 50%

```

SELECT TOP 100000 u.* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY u.DisplayName
Missing Index (Impact 97.5955): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON
    (Users) FOR Sort Cost: 0 %

```

Query 2: Query cost (relative to the batch): 50%

```

SELECT TOP 100000 u.* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY u.DisplayName
Missing Index (Impact 97.5955): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON
    (Users) FOR Sort Cost: 0 %

```

## 2019? Bad news.

@Rep = 1 still gets the tiny data plan – that's fair, MS didn't say they fixed that.

But it also gets the tiny grant at first, as you can see by hovering over the sort to measure spills.

3.4 p52

# IT GOT WORSE

Spills 167,303 pages to  
TempDB!

But hang on: this is where it gets better – run it again with `@Rep = 1`.

3.4 p53

```

171 EXEC usp_UsersByReputation @Reputation = 2;
172 GO
173 EXEC usp_UsersByReputation @Reputation = 1;
174 GO
175 EXEC usp_UsersByReputation @Reputation = 1;
176 /* Adaptive just kicked in, yo */

150 % < >
Results Messages Execution plan
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100000 u.* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY
Missing Index (Impact 97.5955): CREATE NONCLUSTERED INDEX [<Name of Missing Inde:

```

```

    graph LR
        Sort[Sort (Top N Sort)] --> Join[Nested Loops (Inner Join)]
        IndexSeek[Index Seek (NonClustered)] --> Join
        Join --> KeyLookup[Key Lookup (Clustered)]

```

Execution plan details:

- Sort (Top N Sort)**: Cost: 2 %, 26.928ms, 100000 of 9149 (66067%)
- Index Seek (NonClustered)**: Cost: 0 %, 9.192ms, 6044557 of 9149 (66067%)
- Key Lookup (Clustered)**: Cost: 98 %, 6.554ms, 6044557 of 9149 (66067%)

## No spills!

Adaptive memory grants started raising the memory grant.

3.4 p54

## How it works

After a query finishes, SQL reviews its memory use.

- If it spilled to disk, it didn't get enough RAM.  
Raise its desired memory grant for next time.
- If it left a lot of RAM unused, it got too much.  
Lower its desired memory grant for next time.



## How it works, translated

The amount of memory your query gets is based on the parameters that the last person used.

If the last person asked for big data, you get big RAM.

If they asked for small data, you get small RAM.



3.4 p56

## So what's gonna happen now?

I'm about to run it for reputation = 2. What happens?

```
169  DBCC FREEPROCCACHE;
170  GO
171  EXEC usp_UsersByReputation @Reputation = 2;
172  GO
173  EXEC usp_UsersByReputation @Reputation = 1;
174  GO
175  EXEC usp_UsersByReputation @Reputation = 1;
176  /* Adaptive just kicked in, yo */
177  GO
178  EXEC usp_UsersByReputation @Reputation = 2;
179  GO
```



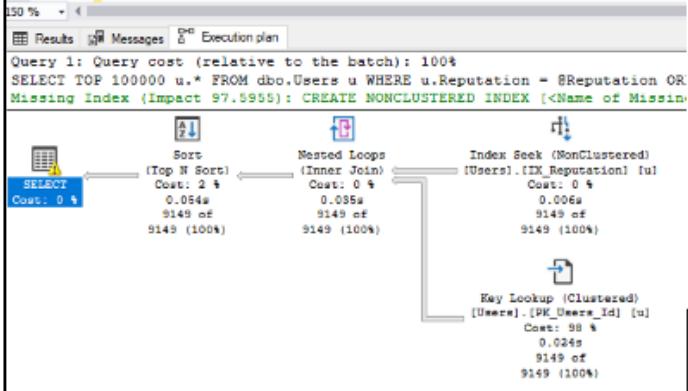
3.4 p57

```

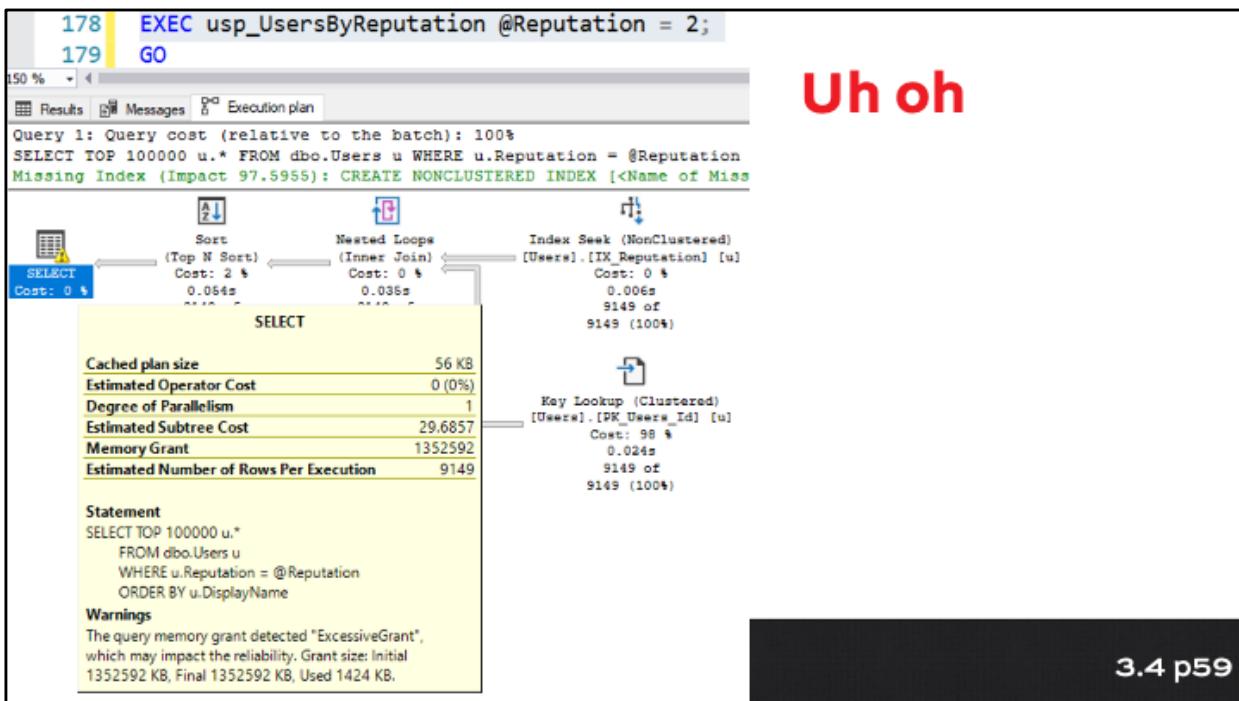
169 DBCC FREEPROCCACHE;
170 GO
171 EXEC usp_UsersByReputation @Reputation = 2;
172 GO
173 EXEC usp_UsersByReputation @Reputation = 1;
174 GO
175 EXEC usp_UsersByReputation @Reputation = 1;
176 /* Adaptive just kicked in, yo */
177 GO
178 EXEC usp_UsersByReputation @Reputation = 2;
179 GO

```

**Runs fast...**



3.4 p58



## You see this coming, don't you?

```
169  DBCC FREEPROCCACHE;
170  GO
171  EXEC usp_UsersByReputation @Reputation = 2;
172  GO
173  EXEC usp_UsersByReputation @Reputation = 1;
174  GO
175  EXEC usp_UsersByReputation @Reputation = 1;
176  /* Adaptive just kicked in, yo */
177  GO
178  EXEC usp_UsersByReputation @Reputation = 2;
179  GO
180  EXEC usp_UsersByReputation @Reputation = 1;
181  GO
```



3.4 p60

```

169 DBCC FREEPROCCACHE;
170 GO
171 EXEC usp_UsersByReputation @Reputation = 2;
172 GO
173 EXEC usp_UsersByReputation @Reputation = 1;
174 GO
175 EXEC usp_UsersByReputation @Reputation = 1;
176 /* Adaptive just kicked in, yo */
177 GO
178 EXEC usp_UsersByReputation @Reputation = 2;
179 GO
180 EXEC usp_UsersByReputation @Reputation = 1;
181 GO

```

Execution plan:

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

SELECT TOP 100000 u.\* FROM dbo.Users u WHERE u.Reputation = @Reputation ORDER BY Missing Index (Impact 97.5955); CREATE NONCLUSTERED INDEX [Name of Missing Index]

```

graph TD
    A[SELECT] --> B[Sort]
    B --> C[Nested Loops]
    C --> D[Index Seek]
    D --> E[Key Lookup]

```

**3.4 p61**

## Back to spilling

The “adaptive” grant doesn’t actually remember anything – it just adapts to the LAST person’s params.

And how big is the spill?

```

169 DBCC FREEPROCCACHE;
170 GO
171 EXEC usp_UsersByReputation @Reputation = 2;
172 GO
173 EXEC usp_UsersByReputation @Reputation = 1;
174 GO
175 EXEC usp_UsersByReputation @Reputation = 1;
176 /* Adaptive just kicked in, yo */
177 GO
178 EXEC usp_UsersByReputation @Reputation = 2;
179 GO
180 EXEC u
181 GO

```

Sort the input.

Physical Operation	Sort
Logical Operation	Top N Sort
Actual Number of Rows for All Executions	100000
Actual Number of Batches	0
Estimated Operator Cost	0.0664 (2%)
Estimated I/O Cost	0.0112613
Estimated CPU Cost	0.485171
Estimated Subtree Cost	29.6837
Number of Executions	1
Estimated Number of Executions	9149
Estimated Row Size	4446.8
Actual Rewinds	0
Actual Rewinds	0
Node ID	0

**Output List**

- [StackOverflow].[dbo].[Users].Id, [StackOverflow].[dbo].[Users].Name, [StackOverflow].[dbo].[Users].CreationDate, [StackOverflow].[dbo].[Users].DisplayName, [StackOverflow].[dbo].[Users].EmailHash, [StackOverflow].[dbo].[Users].LastAccessDate, [StackOverflow].[dbo].[Users].Location, [StackOverflow].[dbo].[Users].Reputation, [StackOverflow].[dbo].[Users].Views, [StackOverflow]...

**Warnings**

Operator used tempdb to spill data during execution with spill level 2 and 1 spilled threads(s). Sort wrote 167115 pages to and read 167115 pages from tempdb with granted memory 2136KB and used memory 2136KB

**Order By**

[StackOverflow].[dbo].[Users].DisplayName Ascending

Query executed successfully.

# 167,115 pages.

Worse than it was before.

3.4 p62



<sigh>



3.4 p63

	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Memory grant feedback (adaptive grants)</b>										
For columnstore indexes	-	Yes	Enterprise	140	Yes	-	Kinda	Yes	-	OK
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	Yes	Test it hard

Adaptive grants for rowstore are rough right now:

- No proactive warning when it's happening
- Minimal instrumentation
- Monitoring tools don't cover it
- Resets constantly on index rebuilds, stats updates, failovers, restarts

Yet they're on by default in compat 150.



3.4 p64

## Because I know you're gonna ask

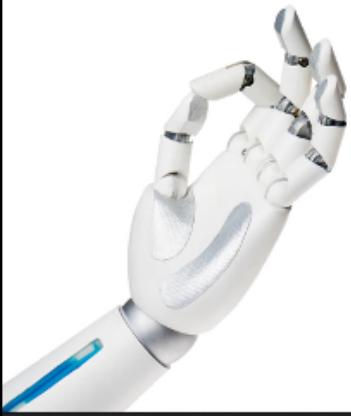
```
ALTER DATABASE SCOPED CONFIGURATION  
SET BATCH_MODE_MEMORY_GRANT_FEEDBACK = OFF;
```

```
ALTER DATABASE SCOPED CONFIGURATION  
SET ROW_MODE_MEMORY_GRANT_FEEDBACK = OFF;
```



3.4 p65

# Adaptive Joins



3.4 p66

## Adaptive join example

You need to join between two tables.  
(Not 2 indexes: this doesn't fix key lookups.)

When the driver table returns just a few rows,  
then it makes sense to do nested loops:  
one operation per row that it finds.

When the driver table returns a lot of rows,  
then it makes sense to do a hash join:  
scan the other table and check all the rows.



3.4 p67

## Sample query

```
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 150; /* SQL Server 2019 */
GO
CREATE OR ALTER PROC dbo.usp_UsersByReputation @Reputation INT AS
    SELECT TOP 100000 u.Id, p.Title, p.Score
        FROM dbo.Users u
        JOIN dbo.Posts p ON p.OwnerUserId = u.Id
        WHERE u.Reputation = @Reputation
        ORDER BY p.Score DESC;
GO

/* And run it: */
EXEC usp_UsersByReputation @Reputation = 1;
GO
```



3.4 p68

```

198 ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 150; /* SQL Server 2019 */
199 GO
200 CREATE OR ALTER PROC dbo.usp_UsersByReputation @Reputation INT AS
201     SELECT TOP 100000 u.Id, p.Title, p.Score
202         FROM dbo.Users u
203             JOIN dbo.Posts p ON p.OwnerUserId = u.Id
204                 WHERE u.Reputation = @Reputation
205                     ORDER BY p.Score DESC;
206 GO

```

1: the robot knows output rows might vary.

2: so nested loops OR hash.

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

SELECT TOP 100000 u.Id, p.Title, p.Score FROM dbo.Users u JOIN dbo.Posts p ON p.OwnerUserId = u.Id WHERE u.Reputation = @Reputation

3.4 p69

## Both operations are shown, but only one will be executed.

```
sts p ON p.OwnerUserId = u.Id WHERE u.Reputation = @Reputation
```

The execution plan details the following steps:

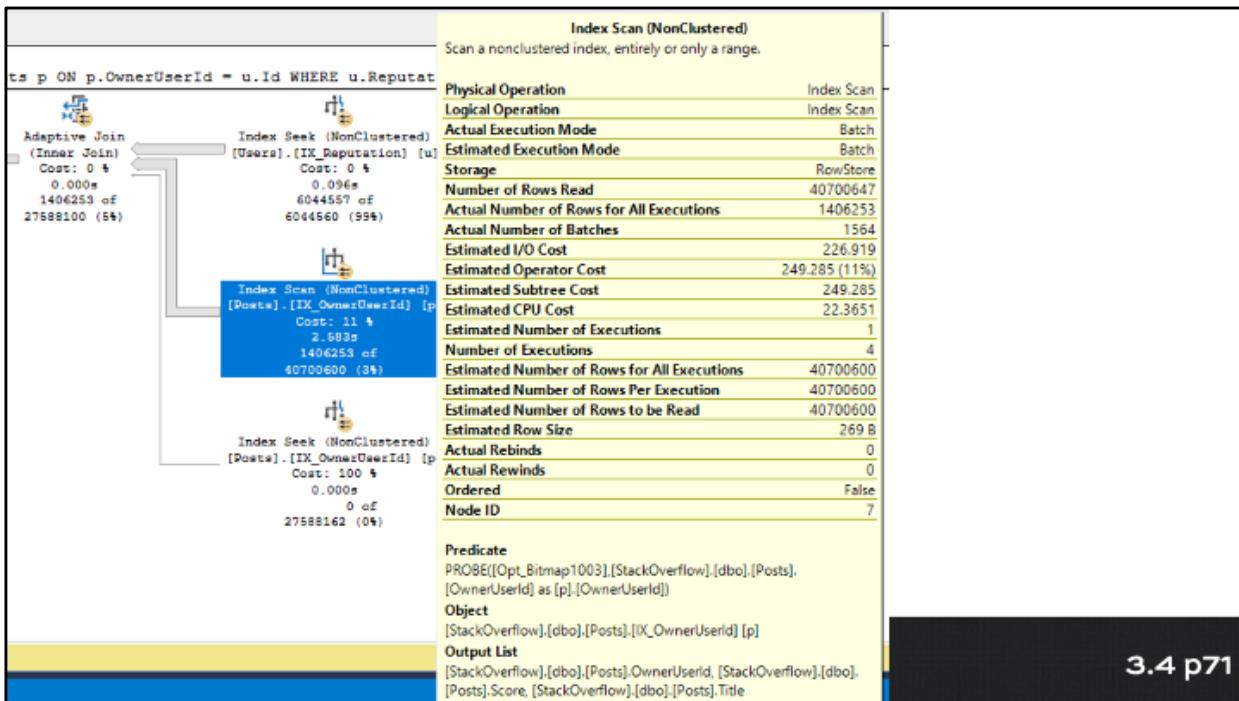
- Adaptive Join [Inner Join]**:
  - Cost: 0 \$
  - 0.00ms
  - 1406288 of 27588100 (5%)
- Index Seek (NonClustered) [Users].[IX\_Reputation] [u]**:
  - Cost: 0 \$
  - 0.00ms
  - 6044560 of 6044560 (99%)
- Index Scan (NonClustered) [Posts].[IX\_OwnerUserId] [p1]**:
  - Cost: 11 \$
  - 2.0ms
  - 1406288 of 40700600 (3%)
- Index Seek (NonClustered) [Posts].[IX\_OwnerUserId] [p1]**:
  - Cost: 100 %
  - 0.00ms
  - 0 of 27588162 (0%)

Hash? Then we'll scan all the data.

Nested loops? We'll just seek a few rows.



3.4 p70



```

221 Try another reputation, and it chooses the seek: */
222 EXEC usp_UsersByReputation @Reputation = 2;
223 GO

```

150 %

Results Messages Execution plan

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

SELECT TOP 100000 u.Id, p.Title, p.Score FROM dbo.Users u JOIN dbo.Posts p ON p.OwnerUserId = u.Id WHERE u.Reput

The execution plan diagram illustrates the query flow. It starts with a **SELECT** node (Cost: 0 %) which feeds into a **Top** node (Cost: 0 %). This is followed by a **Parallelism** node (Cost: 0 %), a **(Gather Streams)** node (Cost: 0 %), and a **Sort** node (Cost: 00 %). The **Sort** node then feeds into an **Adaptive Join** node (Cost: 0 %), which performs an **(Inner Join)**. Finally, the result of the join is used in an **Index Seek (NonClustered)** node (Cost: 0 %) on the **[Users].[IX\_Reputation] [u]** index. The seek operation retrieves 9149 rows from a total of 6044560.

**Try reputation 2,  
and the seek runs.**

72

## Adaptive joins are rarely seen.

In 2017: requires a columnstore index, compat 140

In 2019: works with rowstore indexes, compat 150

Requires batch mode execution (which in 2017 means a columnstore index somewhere in your query, anywhere, even an empty one)

SELECT queries only (no DUILs)

A join that can work with nested loops or hash



3.4 p73

	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Adaptive joins</b>										
For plans with columnstore indexes	Yes	Yes	Enterprise	140	Yes	-	Yes	Yes	-	OK
For all plans	-	Yes	Enterprise	150	-	Yes	Yes	Yes	-	OK

**Adaptive Joins are OK.**

They're really hard to get today, and by themselves, they don't usually get you across the finish line with no other query changes.

But they're OK.





3.4 p74

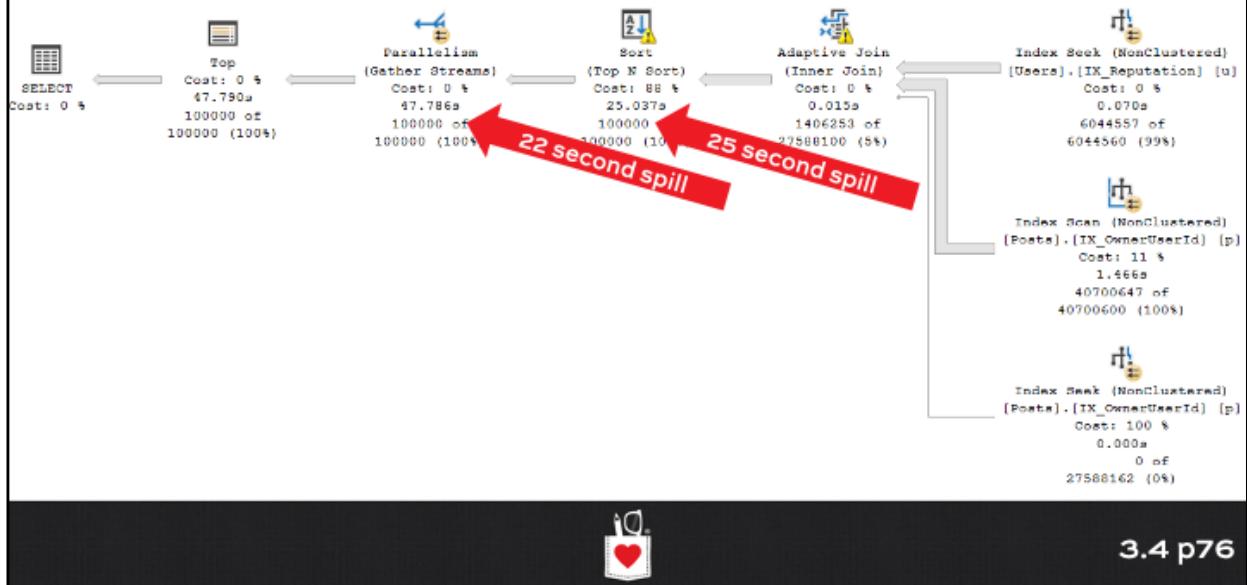
## But run them back to back...

Action	Duration
DBCC FREEPROCCACHE	
Run it for @Reputation = 1	1 second
Run it for @Reputation = 2	0 seconds
Run it for @Reputation = 1 again	<b>1 minute 15 seconds</b>
Why do you suppose that is?	



3.4 p75

## Ah, our “friend” adaptive grants.



## Parameter sniffing is still hard.

One new feature (adaptive grants)  
can wreak havoc on other new features.

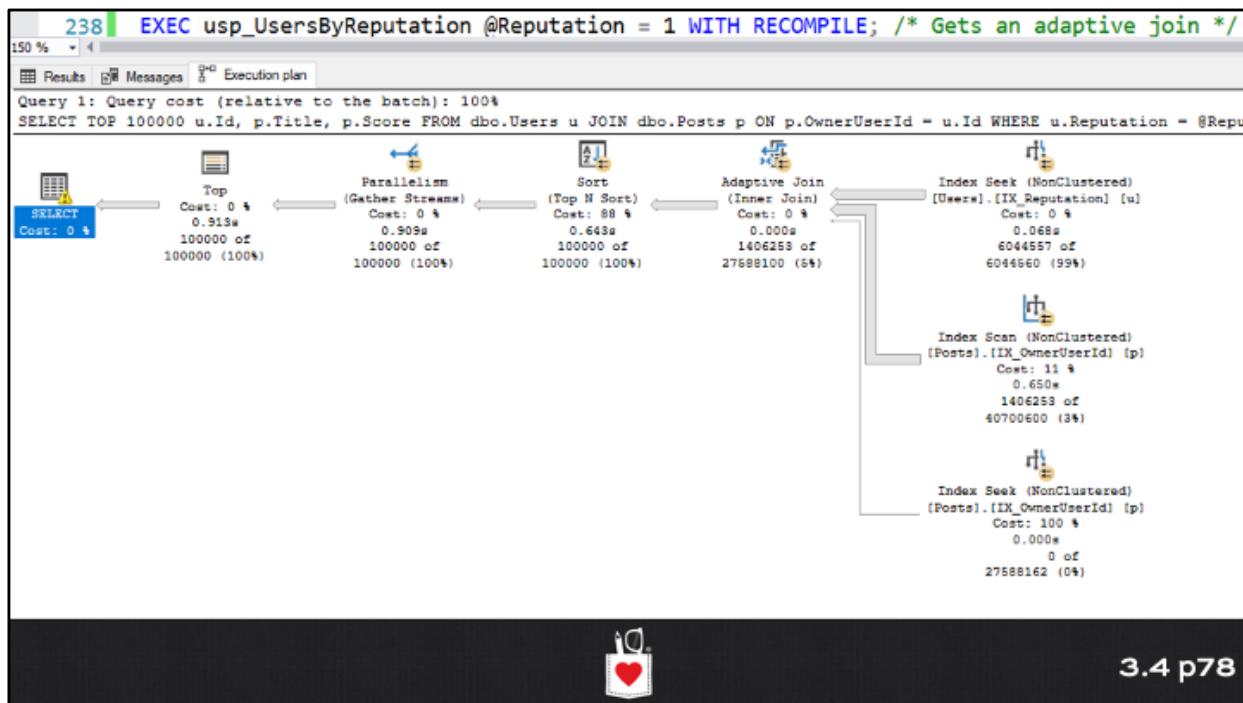
This is called “feature interoperability.”

You’re lucky if new features are tested in isolation,  
let alone combined with other features.

Plus, we still have good ol’ parameter sniffing:  
adaptive joins may not show up for all params.



3.4 p77



3.4 p78

```
239 EXEC usp_UsersByReputation @Reputation = 2 WITH RECOMPILE;
240 /* Index seek on Users, no adaptive join, single threaded */
150 % < Execution plan
```

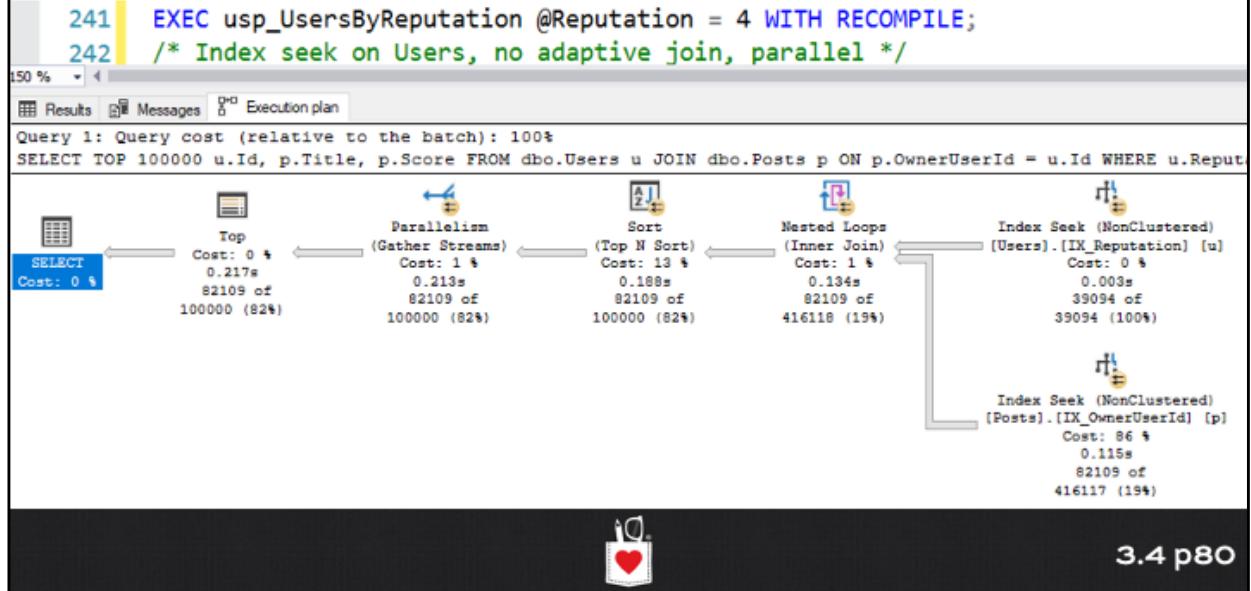
Results Messages Execution plan

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

```
SELECT TOP 100000 u.Id, p.Title, p.Score FROM dbo.Users u JOIN dbo.Posts p ON p.OwnerUserId
```

The execution plan diagram illustrates the query flow. It starts with a 'SELECT' node (Cost: 0 %) which feeds into a 'Sort' node (Top N Sort, Cost: 20 %). This is followed by a 'Nested Loops' join node (Inner Join, Cost: 1 %). The right side of the join receives data from an 'Index Seek (NonClustered)' node on the [Users].[IX\_Reputation] index (Cost: 0 %). The left side of the join receives data from an 'Index Seek (NonClustered)' node on the [Posts].[IX\_OwnerUserId] index (Cost: 79 %). Both seek nodes have costs of 0.001s and 0.034s respectively, with 25397 output rows each, and 97382 (26%) total rows.

## And there's yet another plan:



## Parameter sniffing is HARD now.

This is a very simple stored procedure, but:

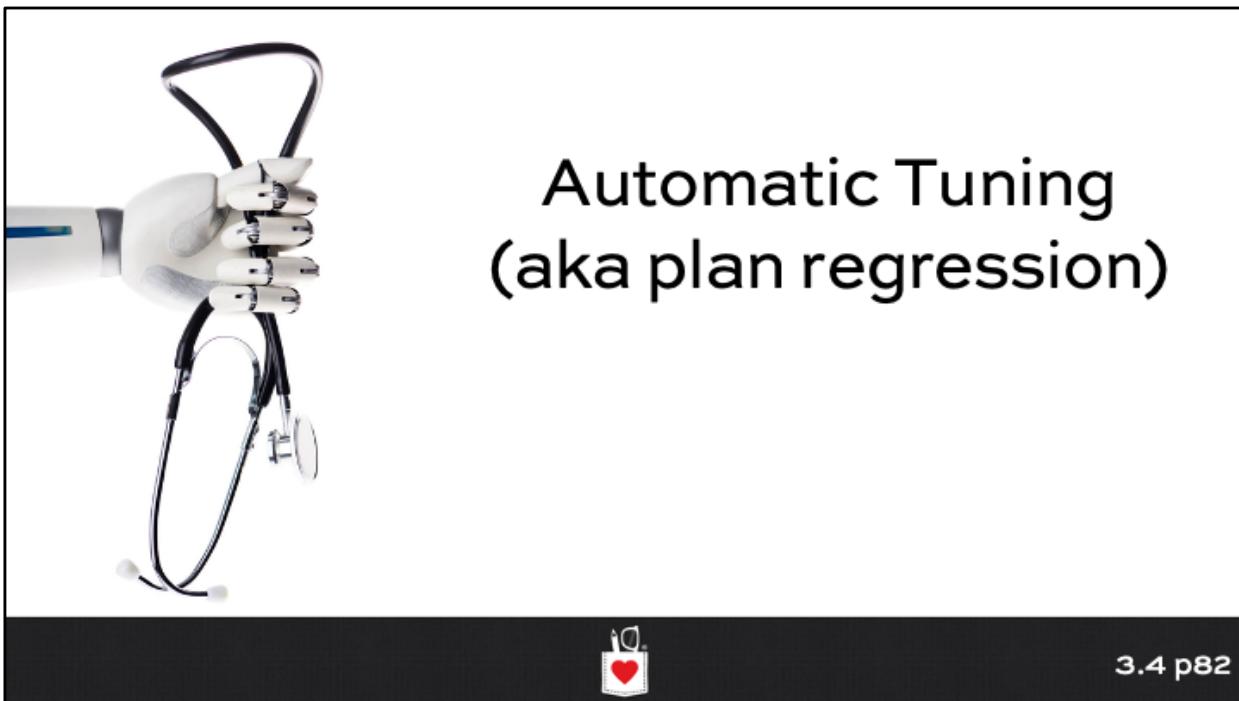
Depending on which @Reputation it's called with first (1, 2, or 4), I can get 3 different plans in cache.

And then every single time it runs, it can set different memory grants for the very next execution.

Plan changes are a huge problem in 2019.



3.4 p81



## This feature name is a flat out lie.

Automatic Tuning implies that SQL Server is actively doing something to make your query better.

It's not even close.



## The sales pitch

Turn on Query Store (which is a story to begin with.)

Get a good query plan.

Have something go wrong: stats update, data change.

Get a bad query plan.

After a while, SQL Server will realize it's a bad plan,  
and try to revert to the good plan that was stored in  
Query Store. If it's faster, keep it permanently.



3.4 p84

## Scenarios it doesn't help

If you never get a good plan in the first place

If you didn't have Query Store already enabled when the good plan was in effect

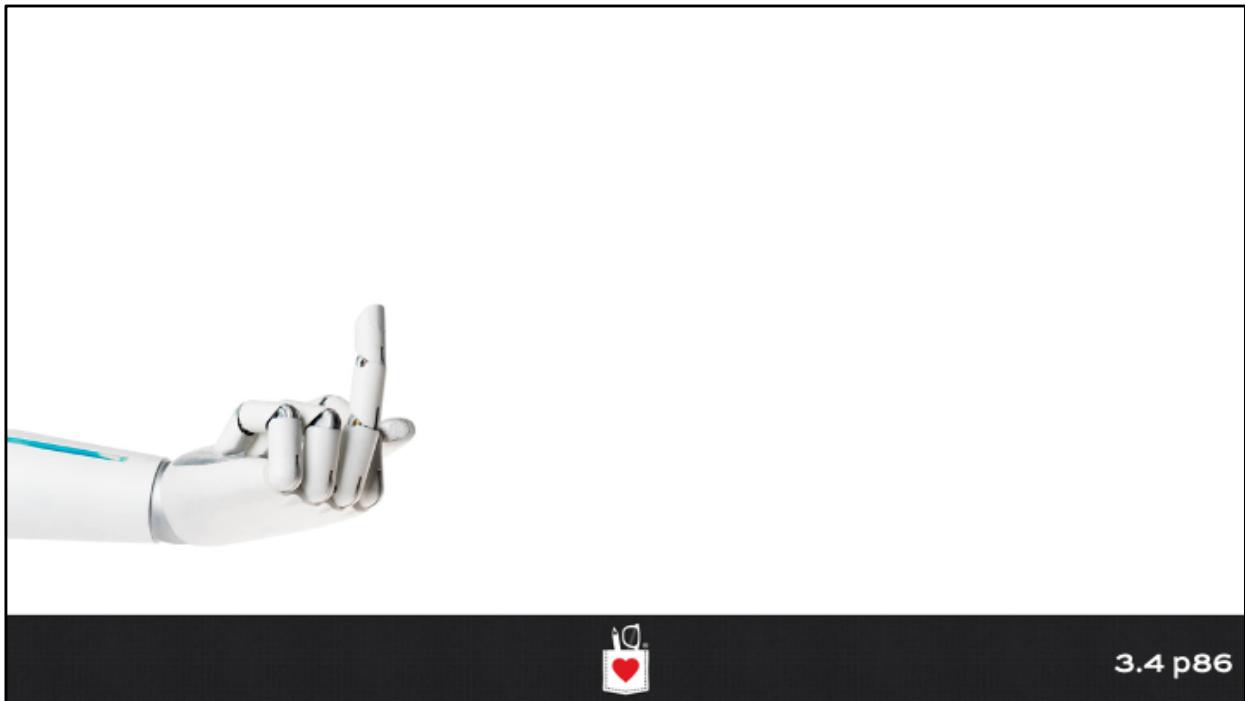
If the query changes (even by one space, like if you do a deployment and just changed a comment)

If there's no one good plan for all parameters (common)

This is not Automatic Tuning  
in any way, shape, or form.



3.4 p85



**Anyway, Microsoft  
is off to a good start.**



3.4 p87

	SQL Server 2017	SQL Server 2019	Minimum Edition	Minimum Compat Level	Works in Azure SQL DB	T-SQL Changes Required	Can be seen in plan	Can make things better	Can make things much worse	Brent's Verdict
<b>Automatic tuning</b>										
Query Store automatic plan correction	Yes	Yes	Enterprise	140	Yes	-	-	Yes	-	OK
Automatic index management	-	-	-	140	Yes	-	-	Yes	-	OK
<b>Batch mode execution</b>										
For columnstore indexes	Yes	Yes	Standard	140	Yes	-	Kinda	Yes	-	Great!
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	-	Great!
<b>Functions</b>										
TVF interleaved execution	Yes	Yes	Standard	140	Yes	-	Yes	Yes	-	OK
Scalar function inlining	-	Yes	Standard	150	-	-	Yes	Yes	Yes	Test it hard
<b>Memory grant feedback (adaptive grants)</b>										
For columnstore indexes	-	Yes	Enterprise	140	Yes	-	Kinda	Yes	-	OK
For rowstore indexes	-	Yes	Enterprise	150	Yes	-	Kinda	Yes	Yes	Test it hard
<b>Adaptive joins</b>										
For plans with columnstore indexes	Yes	Yes	Enterprise	140	Yes	-	Yes	Yes	-	OK
For all plans	-	Yes	Enterprise	150	-	-	Yes	Yes	-	OK
<b>Others</b>										
Approximate Count Distinct	-	Yes	Standard	150	Yes	Yes	Yes	Yes	-	OK
Table variable deferred compilation	-	Yes	Standard	150	Yes	-	Yes	Yes	-	OK



3.4 p88

## I have high hopes. Done right:

This can reduce Microsoft's hardware spend in Azure.

They might cut hosting costs to be more competitive with Amazon RDS Aurora MySQL & PostgreSQL.

This can give Microsoft a very competitive advantage over other on-premises databases, too.

Done right, there's really no downside (other than Microsoft's costs to build it.)



3.4 p89

## But today, most require Enterprise Edition.

(And today, I don't have a problem with that because the features are so limited. If/when open source competitors start to eat Microsoft's lunch in the Standard Edition market, they can open up some of these features at lower price tiers.)



3.4 p90