



BRENT OZAR
UNLIMITED®

Why Is The Same Query Sometimes Slow?

Parameter sniffing, and why **OPTION RECOMPILE** isn't your friend.

Video, related articles, and more: BrentOzar.com/go/sniff



BRENT OZAR
UNLIMITED

99-05: dev, architect, DBA
05-08: DBA, VM, SAN admin
08-10: MCM, Quest Software
Since: consulting DBA

www.BrentOzar.com
Help@BrentOzar.com



**Everyone thinks they
know something
about this.**



They're usually wrong. Buckle up.

People often:

- Blame statistics or fragmentation
- Use the wrong code to reproduce an issue

There is no single right answer

- Pros and cons to every option
- Evaluate each instance individually

**It's almost impossible to identify and diagnose this
without access to the production environment**



I want you to learn 3 things

1. What parameter sniffing is
2. How to react to parameter sniffing emergencies
3. 7 options for fixing the problem long-term



The first thing to learn

1. What parameter sniffing is



About my demo setup

SQL Server 2019, latest Cumulative Update

Stack Overflow 2013 database (50GB)
from <https://BrentOzar.com/go/querystack>

SQL Server 2019 compat level

4 CPU cores, 30GB RAM



Build a query plan for the SELECT now.

```
CREATE INDEX Reputation
ON dbo.Users (Reputation);
GO

SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation = 2
ORDER BY DisplayName;
```



Breaking it down

```
SELECT TOP 10000 *
```

Our index doesn't have all the fields, so we may need a key lookup

```
FROM dbo.Users
```

```
WHERE Reputation = 2
```

Will we use the Reputation index?

```
ORDER BY DisplayName;
```

We'll need a memory grant to sort rows after we find 'em



```
48 DBCC SHOW_STATISTICS('dbo.Users', 'Reputation');
```

Name	Updated	Rows	Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1 Reputation	May 10 2022 7:03AM	2465713	2465713	193	0.04430517	8	NO	NULL	2465713

2.5 million Users

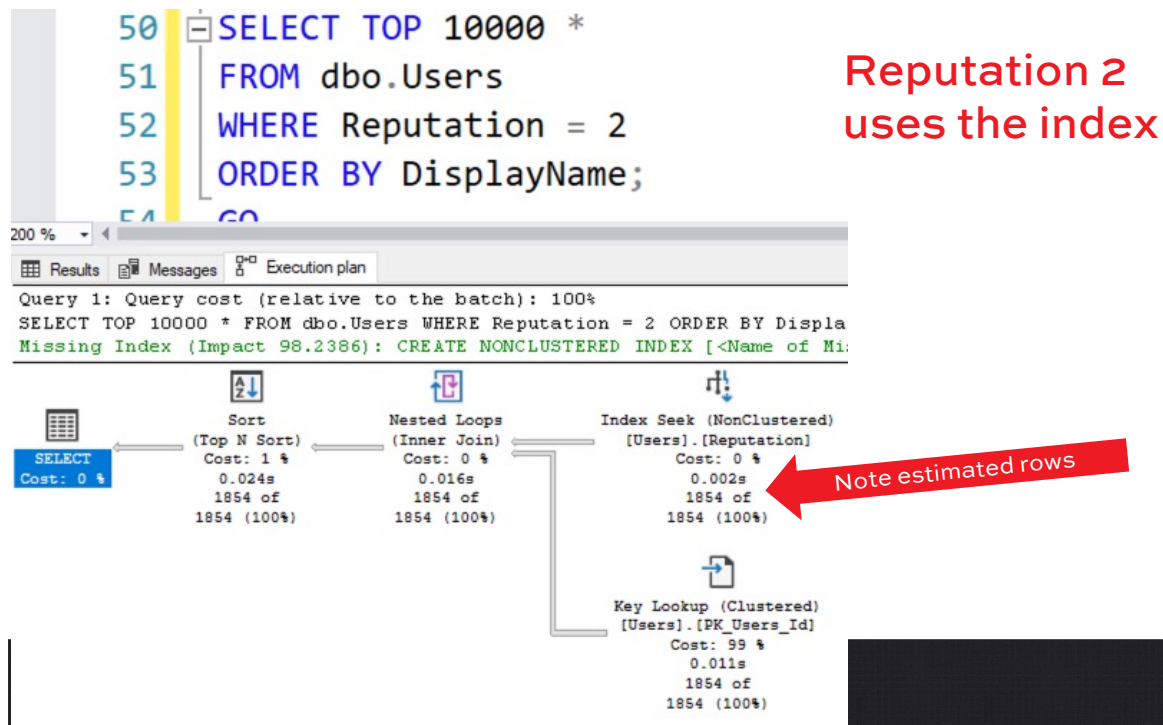
All density	Average Length	Columns
1 5.003252E-05	4	Reputation
2 4.055622E-07	8	Reputation, Id

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	AVG_RANGE_ROWS
1 1	0	1090043	1
2 2	0	1854	1
3 3	0	49987	0
4 4	0	8449	0
5 5	0	7284	0
6 6	0	106467	0

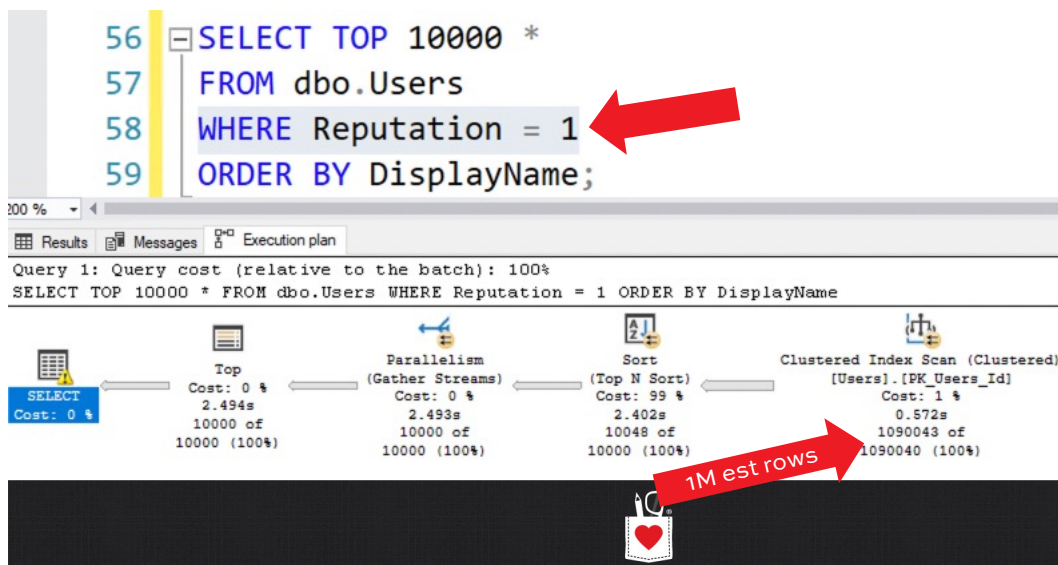
Almost half have just 1 reputation point

Only 1,854 have exactly 2 points





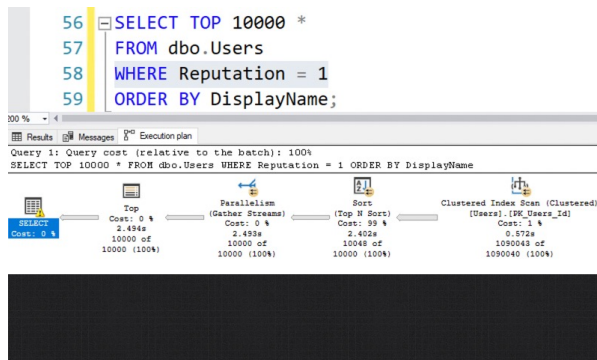
But Reputation 1 ignores the index.



Lookups are expensive

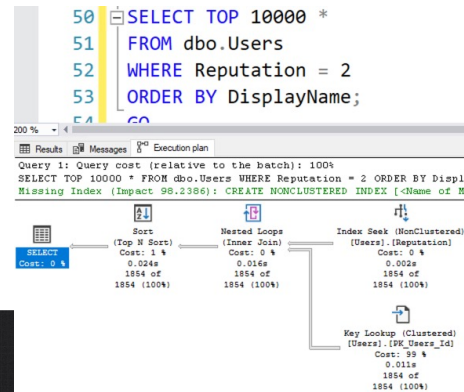
Plan #1

- Preferred when there are **a lot** of rows
- Over SQL Server's tipping point



Plan #2

- Preferred when there are just **a few** rows
- Uses the index, but still gets a request for a covering index



Statistics IO, TIME

Reputation = 1

(1854 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0,
Table 'Users'. Scan count 1, logical reads 5694, p

SQL Server Execution Times:
CPU time = 31 ms, elapsed time = 221 ms.

Uses the index,
uses hardly any
CPU time, doesn't
go parallel.

Reputation = 2

(10000 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0,
Table 'Users'. Scan count 5, logical reads 45020,

SQL Server Execution Times:
CPU time = 6452 ms, elapsed time = 2587 ms.

Scans the clustered
index, does 45020
reads.

Takes ~6 seconds of
CPU time to sort a
lot of rows. Goes
parallel, runs in 3
seconds.

The scorecard

		Plain T-SQL		
Reputation = 2	Duration	0.2 seconds		
	CPU Time	~0s		
	Logical Reads	5,694		
	Parallelism	No		
	Memory grant	1.5MB		
Reputation = 1	Duration	2.6s		
	CPU Time	6.5s		
	Logical Reads	45,020		
	Parallelism	No		
	Memory grant	403MB		



SQL Server uses these literals to pick a plan

```
SELECT * FROM dbo.Users
WHERE Reputation = 2;
GO

SELECT * FROM dbo.Users
WHERE Reputation = 1;
GO
```



SQL Server checks the Reputation = ____ number against your statistics

Index statistics: auto-generated when you create the index

Column statistics:

- Automatically created on the fly when you run queries
- These have the “_WA_Sys” cryptic names

48 DBCC SHOW_STATISTICS('dbo.Users', 'Reputation');

200 %

Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
Reputation	May 10 2022 7:03AM	2465713	2465713	193	0.04430517	8	NO	NULL	2465713

	All density	Average Length	Columns
1	5.003252E-05	4	Reputation
2	4.055622E-07	8	Reputation, Id

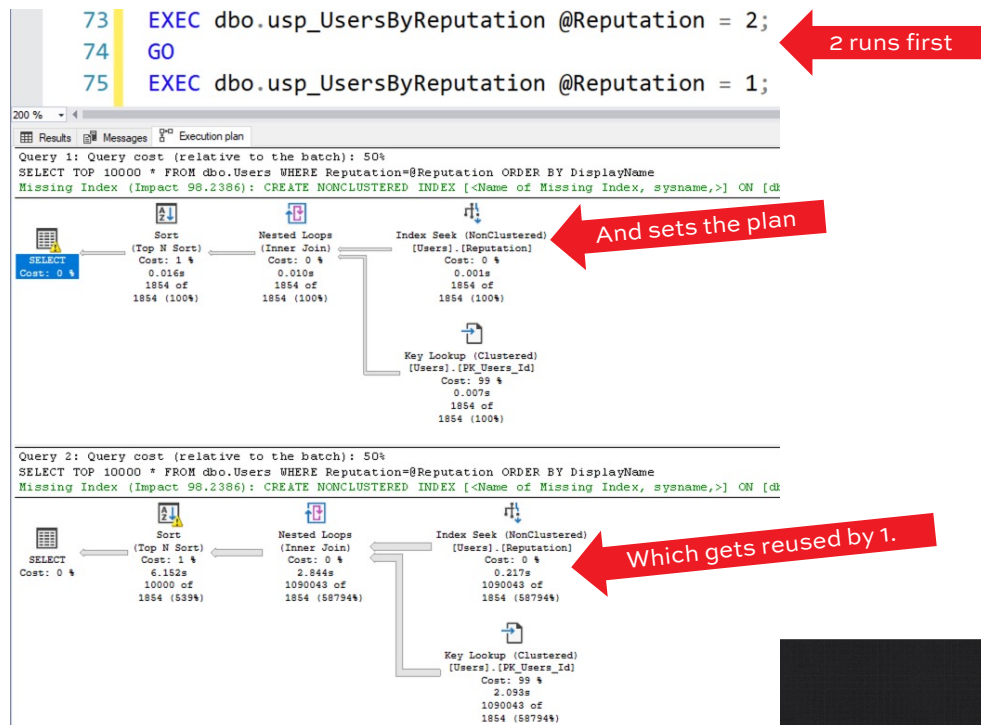
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1090043	0	1
2	2	0	1854	0	1
3	3	0	49987	0	1
4	4	0	8449	0	1
5	5	0	7284	0	1
6	6	0	106167	0	1

Let's move it into a stored procedure

```
CREATE OR ALTER PROCEDURE dbo.usp_UsersByReputation
    @Reputation int
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation = @Reputation
ORDER BY DisplayName;
GO

EXEC dbo.UsersByReputation @Reputation = 2;
GO
EXEC dbo.UsersByReputation @Reputation = 1;
GO
```





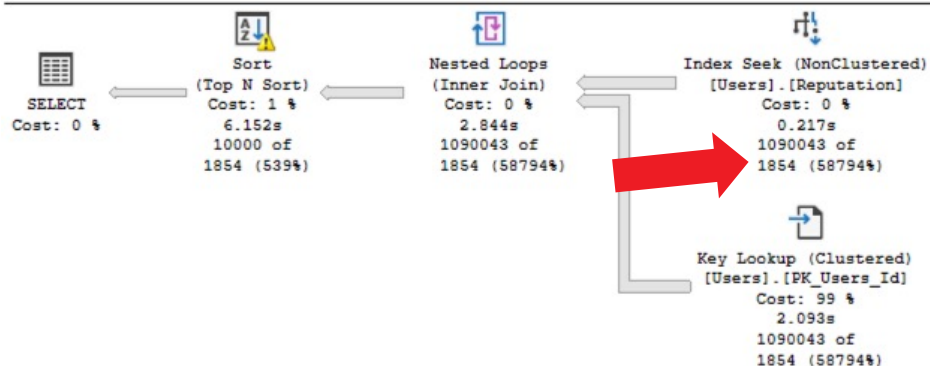
Zoom in on Reputation 1's plan

And note the estimated rows: 1,854. That's from Reputation = 2.

```

SELECT TOP 10000 * FROM dbo.Users WHERE Reputation=@Reputation ORDER BY Dis
Missing Index (Impact 98.2386): CREATE NONCLUSTERED INDEX [<Name of Missing

```



Right click on the SELECT in Rep 1's plan...

Click Properties, and scroll down to Parameters:

⊕	OptimizerHardwareDependentPrope	
⊕	OptimizerStatsUsage	
⊖	Parameter List	@Reputation
	Column	@Reputation
	Parameter Compiled Value	(2)
	Parameter Data Type	int
	Parameter Runtime Value	(1)
	QueryHash	0xΔ6F28073109D7278



The effects are bad.

Reputation = 2

```
Results Messages Execution plan
(1854 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0
Table 'Users'. Scan count 1, logical reads 5694, physical reads 0

SQL Server Execution Times:
    CPU time = 16 ms,  elapsed time = 331 ms.
```

Sets up the index seek plan, runs quickly.

Reputation = 1

```
(10000 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0
Table 'Users'. Scan count 1, logical reads 3340157, physical reads 0

SQL Server Execution Times:
    CPU time = 5906 ms,  elapsed time = 6374 ms.
```

Reuses the seek plan.

Reads more pages than there are in the table.

Only goes single-threaded, slow.



The scorecard

		Plain T-SQL	Reusable Plan, Rep 2 Runs First	Reusable Plan, Rep 1 Runs First
Reputation = 2	Duration	0.2 seconds	0.3 seconds	
	CPU Time	~0s	~0s	
	Logical Reads	5,694	5,694	
	Parallelism	No	No	
	Memory grant	1.5MB	12MB	
Reputation = 1	Duration	2.6s	6.4s	
	CPU Time	6.5s	5.9s	
	Logical Reads	45,020	3,340,157	
	Parallelism	Yes	No	
	Memory grant	403MB	12MB	



What happens if...

Windows restarts

The SQL Server service restarts

Someone runs DBCC FREEPROCCACHE

Indexes on the Users table are rebuilt

Statistics on the Users table are updated

The server comes under memory pressure

And the queries are run in a different order?



```

78 DBCC FREEPROCCACHE
79 GO
80 EXEC dbo.usp_UsersByReputation @Reputation = 1;
81 GO
82 EXEC dbo.usp_UsersByReputation @Reputation = 2;

```

1 runs first

And sets the plan

Which gets reused by 2.

Query 1: Query cost (relative to the batch): 50%
 SELECT TOP 10000 * FROM dbo.Users WHERE Reputation=@Reputation ORDER BY DisplayName

Query 2: Query cost (relative to the batch): 50%
 SELECT TOP 10000 * FROM dbo.Users WHERE Reputation=@Reputation ORDER BY DisplayName

Now we have a different problem.

Reputation = 1

Note: 1 first

(10000 rows affected)
 Table 'Worktable'. Scan count 0, logical reads 0,
 Table 'Users'. Scan count 5, logical reads 45024,

SQL Server Execution Times:
 CPU time = 6750 ms, elapsed time = 3054 ms.

Reputation 1 is back to scanning the table, which is fine...

Reputation = 2

(1854 rows affected)
 Table 'Worktable'. Scan count 0, logical reads 0,
 Table 'Users'. Scan count 5, logical reads 45184,

SQL Server Execution Times:
 CPU time = 657 ms, elapsed time = 473 ms.

But now Reputation 2 is ignoring the index.



The scorecard

		Plain T-SQL	Reusable Plan, Rep 2 Runs First	Reusable Plan, Rep 1 Runs First
Reputation = 2	Duration	0.2 seconds	0.3s	0.5s
	CPU Time	~0s	~0s	0.7s
	Logical Reads	5,694	5,694	45,184
	Parallelism	No	No	Yes
	Memory grant	1.5MB	12MB	405MB
Reputation = 1	Duration	2.6s	6.4s	3s
	CPU Time	6.5s	5.9s	6.8s
	Logical Reads	45,020	3,340,157	45,024
	Parallelism	Yes	No	Yes
	Memory grant	403MB	12MB	5GB



There's no one “good” plan here.

If Reputation 2 runs first, then Reputation 1:

- Does 74x more logical reads
- Takes 2 longer

If Reputation 1 runs first, then Reputation 2:

- Ignores indexes
- Does 8x more logical reads

And in either case, depending on your version & compat level, you can see huge memory grant swings, spills to TempDB, etc.





Next up, let's just try not to freak out

2. How to respond to parameter sniffing emergencies



The career progression of a perf tuner

“Let’s restart Windows!”

“Let’s restart the SQL Server service!”

“Let’s run DBCC FREEPROCCACHE!”

“Let’s rebuild all the indexes!”

“Let’s rebuild the indexes on the one table involved!”

“Let’s update statistics on the table!”

“Let’s just clear this one plan from cache.”



How to free one plan from cache

Run `sp_BlitzCache @ExpertMode = 1`

Find the query that’s not usually on the suckerboard,
because he probably just got an unusually bad plan today

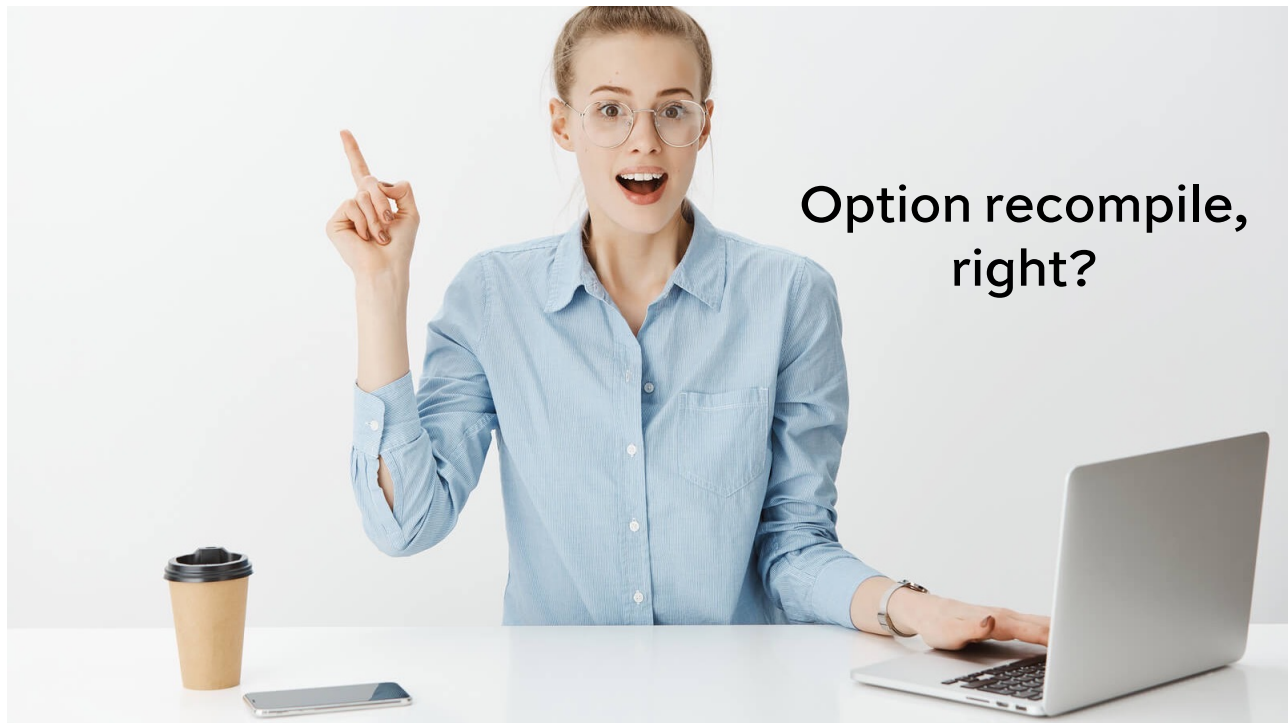
Scroll to the far right, and check out:
DBCC FREEPROCCACHE (mycrappysqlhandle)

Save the execution plan, then free it from cache.

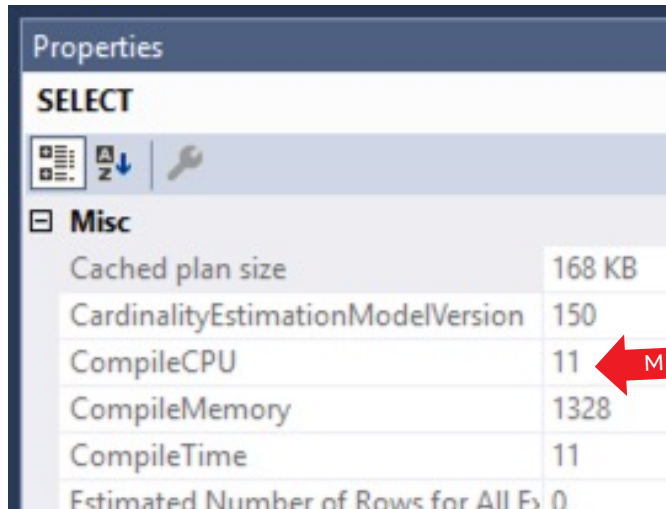
Did the emergency stop? If so, you’ve found your culprit.

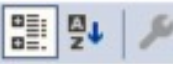


3. 7 ways to fix it for good



It takes CPU time to compile a plan.

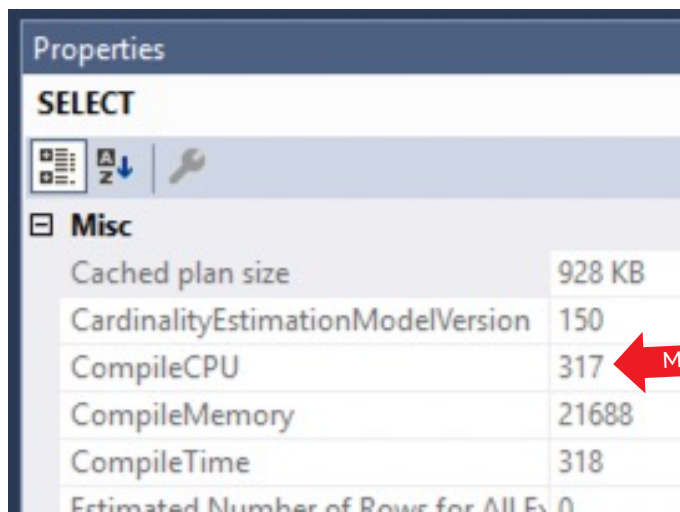


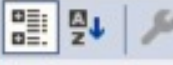
Properties	
SELECT	
	
Misc	
Cached plan size	168 KB
CardinalityEstimationModelVersion	150
CompileCPU	11
CompileMemory	1328
CompileTime	11
Estimated Number of Rows for All Ex 0	

Milliseconds



The more complex the query...



Properties	
SELECT	
	
Misc	
Cached plan size	928 KB
CardinalityEstimationModelVersion	150
CompileCPU	317
CompileMemory	21688
CompileTime	318
Estimated Number of Rows for All Ex 0	

Milliseconds



Math problem

If you want to run this many queries per second:	10
And they each take this many MS to compile:	300
Then you need this much CPU power just to compile:	$10 * 300 = \mathbf{3,000\ ms}$

That's 3 CPU cores just for compiles.

And it doesn't even include the CPU time to **run** the queries.

And that's only 10 queries.



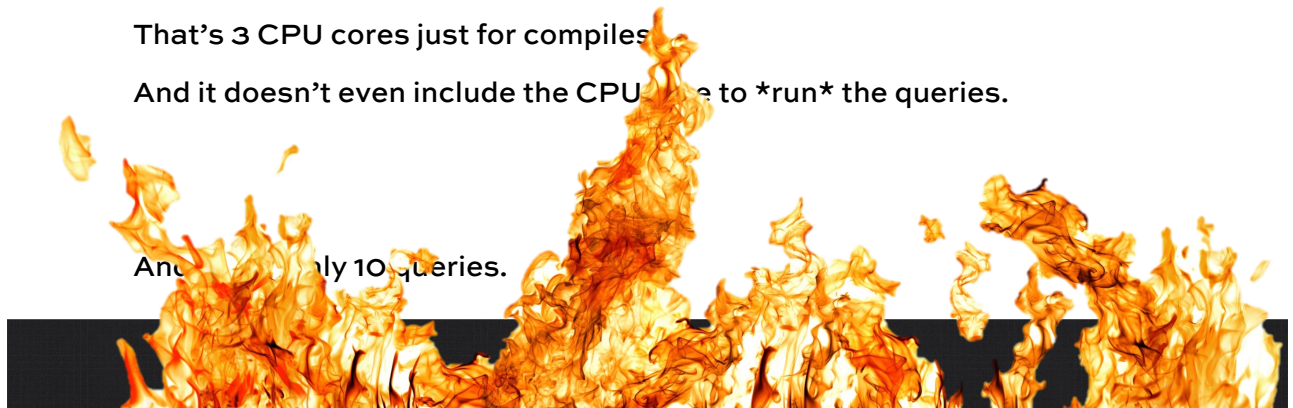
Math problem

If you want to run this many queries per second:	10
And they each take this many MS to compile:	300
Then you need this much CPU power just to compile:	$10 * 300 = \mathbf{3,000\ ms}$

That's 3 CPU cores just for compiles

And it doesn't even include the CPU time to **run** the queries.

And that's only 10 queries.





Yeah. So about that
option recompile.

Now you're waiting to hear
the option that doesn't suck.



All these options suck.

You just have to find the right one
for each parameter sniffing problem,
and every problem is different.



You have a decision to make.

Can you get one good plan for all parameters?

- OPTIMIZE FOR UNKNOWN
- Declare a local variable
- OPTIMIZE FOR specific value
- Plan guides
- Covering indexes

Or do you need different plans for different parameters?

- OPTION RECOMPILE
- Stored procedure branching
- Dynamic SQL comment injection



Option RECOMPILE for the entire proc

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT WITH RECOMPILE
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName;
```



Option RECOMPILE for the entire proc

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT WITH RECOMPILE
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName;
```



* Unless you purposely want to hide your code from many monitoring tools, hee hee.



Option RECOMPILE at the statement level

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName
OPTION (RECOMPILE);
```



Option RECOMPILE at the statement level

Plan cache won't track statement-level query stats
(although it will track stored-proc-level, you just won't know which queries were the expensive ones)

Burns CPU for every execution, compiling a new plan

Makes sense for rarely-run, wildly variable stored procs
(like once every minute or two, max)



Stored procedure branching

```
ALTER PROCEDURE UsersByReputation
    @Reputation INT
AS
IF @Reputation = 1
    EXEC UsersByReputation_1 @1
ELSE
    EXEC UsersByReputation_Other @1
```



Stored procedure branching

Like optimizing for a specific value:

- You'd better know your data pretty well.
- Your data better not change over time.

But now you can have 2 different sub-procedures,
each of which gets its own optimized plan.

They could even have the same code in them:
they'll just get sniffed for the right values, and get the right plans.

(This is a pretty rarely used trick.)



OPTIMIZE FOR UNKNOWN

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName
OPTION (OPTIMIZE FOR UNKNOWN);
GO
```

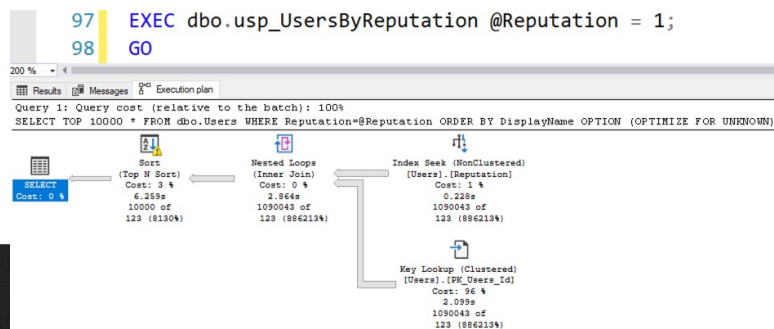


OPTIMIZE FOR UNKNOWN

This forces everyone to use the density vector:
the statistic SQL Server uses for the “average” Reputation.

In some cases, with widely spread-out data, this can work.

In this case, it simply doesn’t work:
the density vector gives us a 123 row estimate.



Fake “OPTIMIZE FOR UNKNOWN”

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
    DECLARE @ReputationUnknown INT;
    SET @ReputationUnknown = @Reputation;

    SELECT TOP 10000 *
    FROM dbo.Users
    WHERE Reputation = @ReputationUnknown
    ORDER BY DisplayName;
GO
```



Fake “OPTIMIZE FOR UNKNOWN”

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
    DECLARE @ReputationUnknown INT;
    SET @ReputationUnknown = @Reputation;

    SELECT TOP 10000 *
    FROM dbo.Users
    WHERE Reputation = @ReputationUnknown
    ORDER BY DisplayName;
GO
```



Fake “OPTIMIZE FOR UNKNOWN”

Exact same results as Optimize For Unknown

The “Optimize for Unknown” hint was added in SQL Server 2008

Prior to that, people did the “create another variable” trick sometimes

It has the same downsides:

- Often just not a very good execution plan
- May not be a stable plan over time

But you look like an idiot,
and people overwrite your code.



If you want the density vector, use this instead.

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName
OPTION (OPTIMIZE FOR UNKNOWN) ;
GO
```



OPTIMIZE FOR Value

```
ALTER PROCEDURE dbo.UsersByReputation
    @Reputation INT
AS
SELECT TOP 10000 *
FROM dbo.Users
WHERE Reputation=@Reputation
ORDER BY DisplayName
OPTION (OPTIMIZE FOR @Reputation = 1)
```



OPTIMIZE FOR Value

You'd better know your data pretty well.

Your data better not change over time.



Plan guides

You identify a very exact string, and the right query plan for it

You guide the plan: that plan sticks around even after restarts

2016's Query Store makes this point-and-click easy

But it's up to you to:

- Identify the string
- Construct the right query plan
- Know when you should un-pin it
- Make sure it works, and that it stays working



Plan guides fail silently.

Query changes at all? Guide won't work.

Indexes in use disappeared? Guide stops working.

Doesn't fix issues like memory grants:
if the memory ain't there, the guide can't do magic.

I love plan guides to fix emergency problems,
but they're like duct tape.
You probably need a better plan within 1-2 weeks.



Hard solution: better plan for everyone

Most parameter sniffing issues have a simple cause:

- There's no covering index, so
- Sometimes a seek + key lookup is better, but
- Sometimes a clustered index scan is better

You could solve it by creating a better index that fits more scenarios.



What about our demo query's case?

Multi-Plan Solutions	Would it work?
OPTION (RECOMPILE)	Depends on frequency
Stored procedure branching	Yes

Single-Plan Solutions	Would it work?
OPTIMIZE FOR UNKNOWN (or the internal local variable trick)	No, it'd estimate 123 rows here
OPTIMIZE FOR @Reputation = 1	Yes
Plan guide	Yes, but will fail when query changes
Get one good plan for everyone	Clippy's covering index wouldn't work, but a manual index would.





Recap



You learned 3 things

1. What parameter sniffing is
2. How to react to parameter sniffing emergencies:
 - `sp_BlitzCache @ExpertMode = 1`
 - Save the plan
 - Use the “free from cache” columns at the far right
3. 7 options for fixing the problem long-term: there’s no easy button

Learn more: BrentOzar.com/go/sniff

