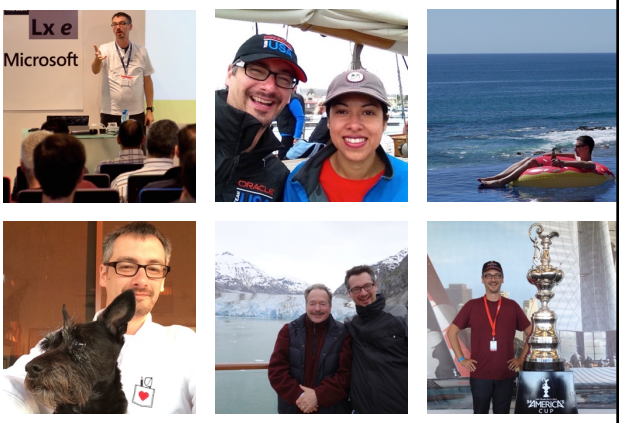
Identifying and Fixing Parameter Sniffing

Brent Ozar

www.BrentOzar.com
sp_Blitz – sp_BlitzFirst
email newsletter – videos
SQL Critical Care®



This is genuinely difficult

It's easy to get confused. People often:

- Blame statistics or fragmentation when they're not the problem
- 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 problem without access to the production environment

The complexity makes this time consuming

I want you to learn 4 things

1. What parameter sniffing is
2. How to react to parameter sniffing emergencies
3. Once you've found a query that's susceptible to parameter sniffing, how to stop testing slow code incorrectly
4. Options for fixing the problem long-term

More details: BrentOzar.com/go/sniff

The first thing to learn

1. What parameter sniffing is

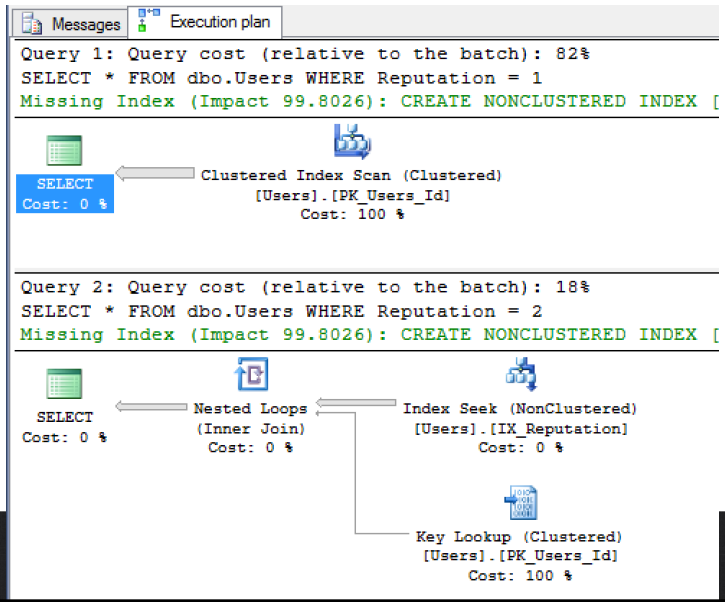
These two queries use literals

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

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

Test: optimize the query

```
/* You have only:
- Clustered PK on Id
- Nonclustered index on Reputation
(no included columns) */

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

Really different execution plans



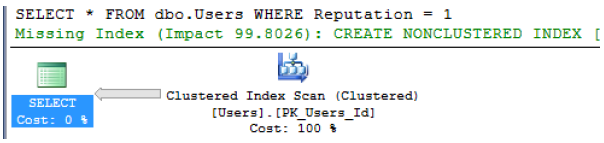
Lookups are expensive

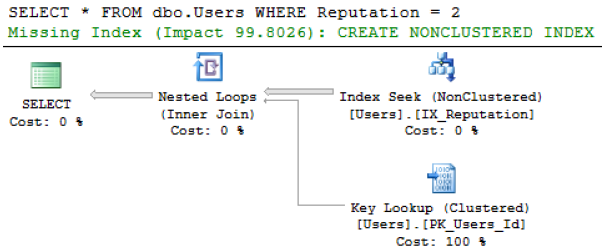
Plan #1

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

Plan #2

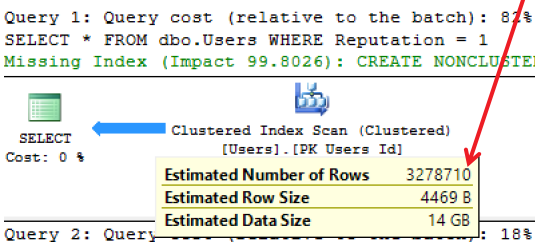
- Preferred when there are just **a few** rows
- Uses the index, but still gets a request for a covering index

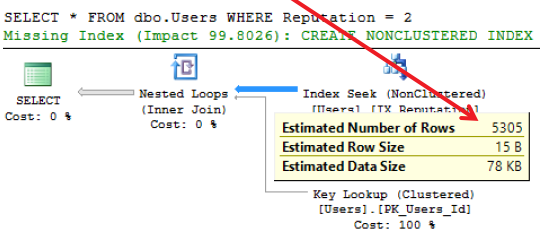




Just like you learned in "How to Think Like the Engine"...

These estimated row counts come from statistics





Two types of statistics

Index statistics

- These are 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

```
CREATE INDEX IX_Reputation ON dbo.Users(Reputation)
GO
DBCC SHOW_STATISTICS('dbo.Users', 'IX_Reputation')
```

100 %

Results | Messages

	Name	Updated	Rows	Rows Sampled	St
1	IX_Reputation	May 23 2016 4:36PM	5277831	5277831	2

	All density	Average Length	Columns
1	5.962674E-05	4	Reputation
2	1.894718E-07	8	Reputation, Id

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_
1	1	0	3278709	0
2	2	0	5305	0
3	3	0	131811	0
4	4	0	24092	0
5	5	0	18802	0
6	6	0	219541	0
7	7	0	9532	0
8	8	0	88122	0
9	9	0	21278	0

Statistics are lightweight

Much smaller than indexes

- Indexes hold data
- Statistics *describe* data

Statistics know

- What the "average rows" are for any given value (the "density vector")
- For the first key column in the statistics, a histogram tracks:
  - How many rows are equal to a distribution of values
  - How many rows are between a value and the prior value

That histogram can have a maximum of 201 steps
(one is for null values)

This does the same thing, right?

```
CREATE PROCEDURE dbo.UsersByReputation
  @Reputation int
AS
SELECT * FROM dbo.Users
WHERE Reputation=@Reputation;
GO

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

Well... no.

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

100 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 50%
SELECT * FROM dbo.Users WHERE Reputation=@Reputation
Missing Index (Impact 99.8026): CREATE NONCLUSTERED INI
```

SELECT
Cost: 0 %

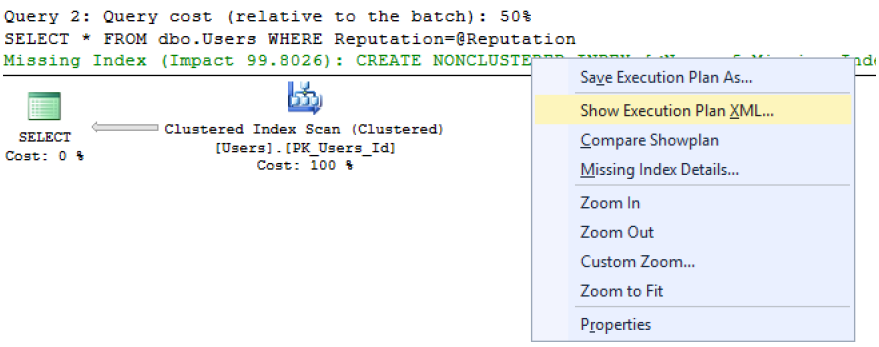
Clustered Index Scan (Clustered)
[Users].[PK_Users_Id]
Cost: 100 %

```
Query 2: Query cost (relative to the batch): 50%
SELECT * FROM dbo.Users WHERE Reputation=@Reputation
Missing Index (Impact 99.8026): CREATE NONCLUSTERED INI
```

SELECT
Cost: 0 %

Clustered Index Scan (Clustered)
[Users].[PK_Users_Id]
Cost: 100 %

Right click on the second execution plan...



Scroll to the bottom of the XML...

Compiled value and runtime value are different

The compiled value came from the first time it was run

This parameter was “sniffed”

```
                </Compare>
              </ScalarOperator>
            </Predicate>
          </IndexScan>
        </RelOp>
        <ParameterList>
          <ColumnReference Column="@Reputation" ParameterCompiledValue="(1)" ParameterRuntimeValue="(2)" />
        </ParameterList>
      </QueryPlan>
    </StmtSimple>
  </Statements>
 </Batch>
</BatchSequence>
</ShowPlanXML>
```

But what happens if...

Windows restarts

The SQL Server service restarts

Someone runs DBCC FREEPROCCACHE

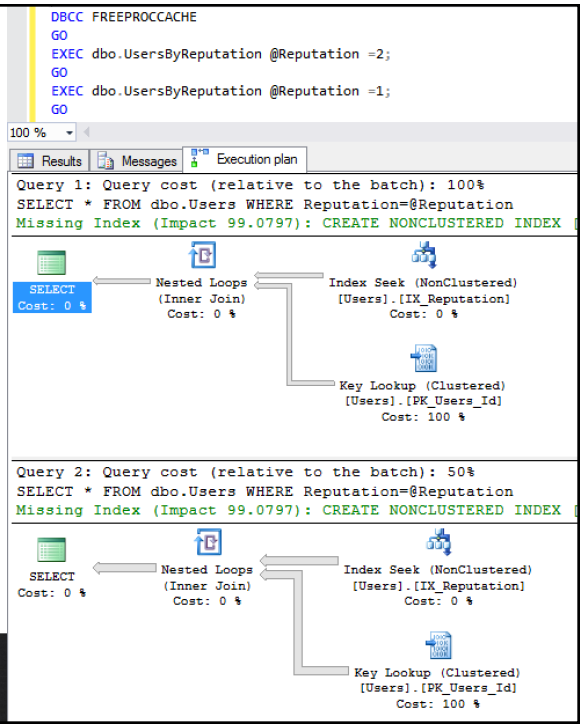
Indexes on the Users table are rebuilt

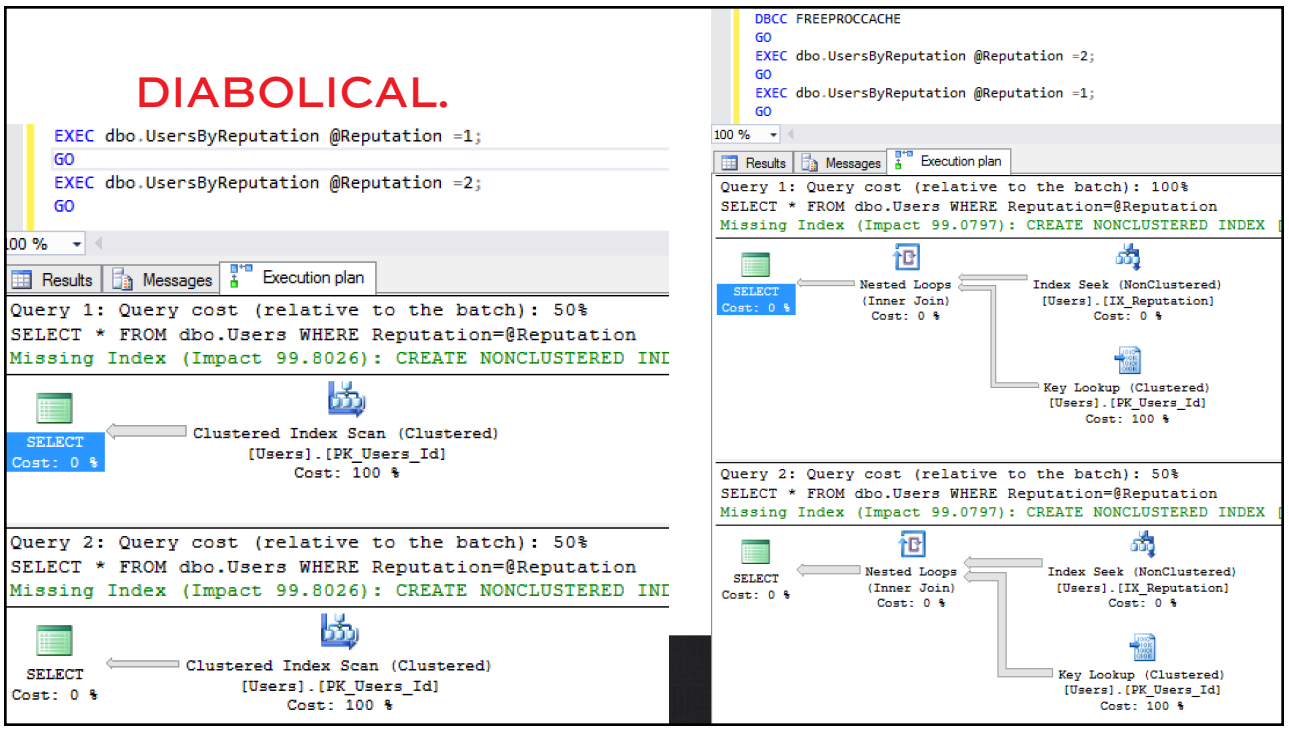
Statistics on the Users table are updated

The server comes under memory pressure

And the queries are run in a different order?

DIABOLICAL.





The problem of parameter sniffing

Say I have a stored procedure called usp_SalesReport that gives me sales for a US state. It has a @State parameter.

Call it with:	And the plan is built for:	Runtime is:
@State = 'TX'	Big states, big data	1 second (great for big states)
@State = 'RI'	Big states, big data	500 milliseconds (which is considered "slow" for tiny states, but who cares?)

The problem of parameter sniffing

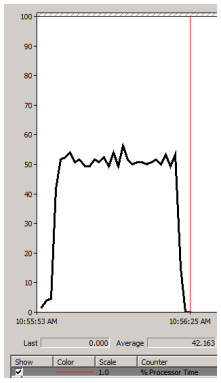
Say I have a stored procedure called usp_SalesReport that gives me sales for a US state. It has a @State parameter.

Call it with:	And the plan is built for:	Runtime is:
@State = 'TX'	Big states, big data	1 second (great for big states)
@State = 'RI'	Big states, big data	500 milliseconds (which is considered "slow" for tiny states, but who cares?)

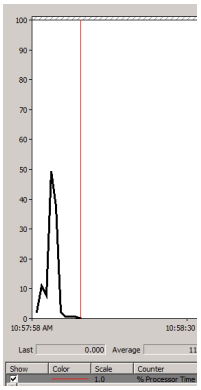
Call it with:	And the plan is built for:	Runtime is:
@State = 'RI'	Tiny states, tiny data	15 milliseconds (great for tiny states)
@State = 'TX'	Tiny states, tiny data	30 seconds (and our server falls over)

Parameter sniffing isn't a bug!

Query that compiles each time – run 5K times



Query using cached plan – run 5K times



% Processor Time on a 2 processor VM. Nothing running but this query.

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.

Just outsource it to StackOverflow

3. How to test vulnerable code

Here's where things go wrong

We want to test out some hints to see if we can make this thing better

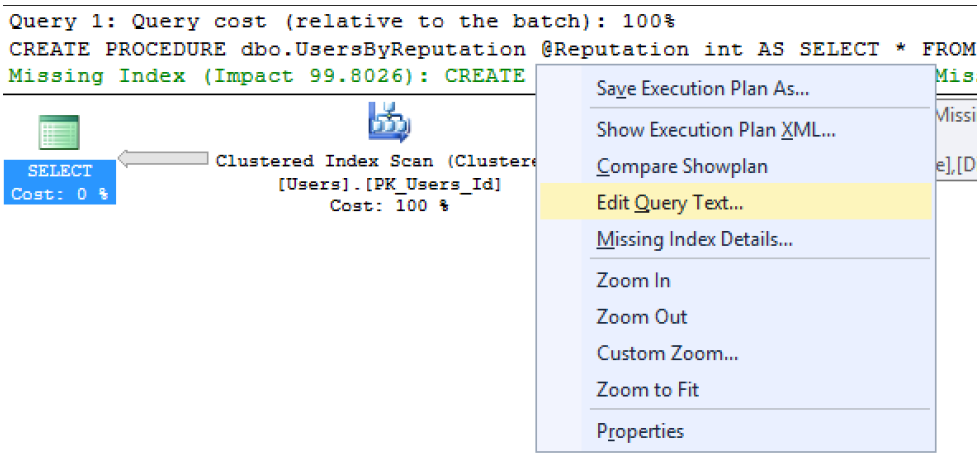
But we don't want to modify the production proc– that'd be crazy

And we want a lightweight way to test

So we re-write this as a simple query

We use a local variable

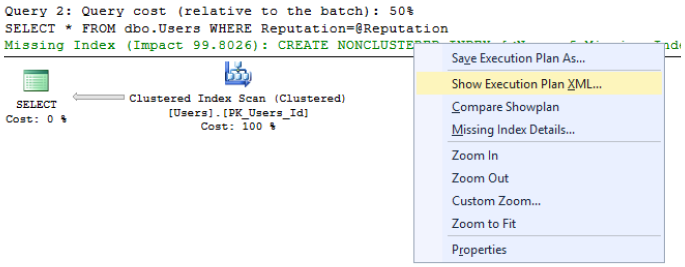
Get the TSQL for the procedure



The production stored procedure code

```
CREATE PROCEDURE dbo.UsersByReputation
  @Reputation int
AS
SELECT * FROM dbo.Users
WHERE Reputation=@Reputation
```

Get the compiled value




```
            </Compare>
          </ScalarOperator>
        </Predicate>
      </IndexScan>
    </RelOp>
    <ParameterList>
      <ColumnReference Column="@Reputation" ParameterCompiledValue="(1)" ParameterRuntimeValue="(2)" />
    </ParameterList>
  </QueryPlan>
 </StmtSimple>
</Statements>
</Batch>
</BatchSequence>
</ShowPlanXML>
```

The WRONG way to test: comment it

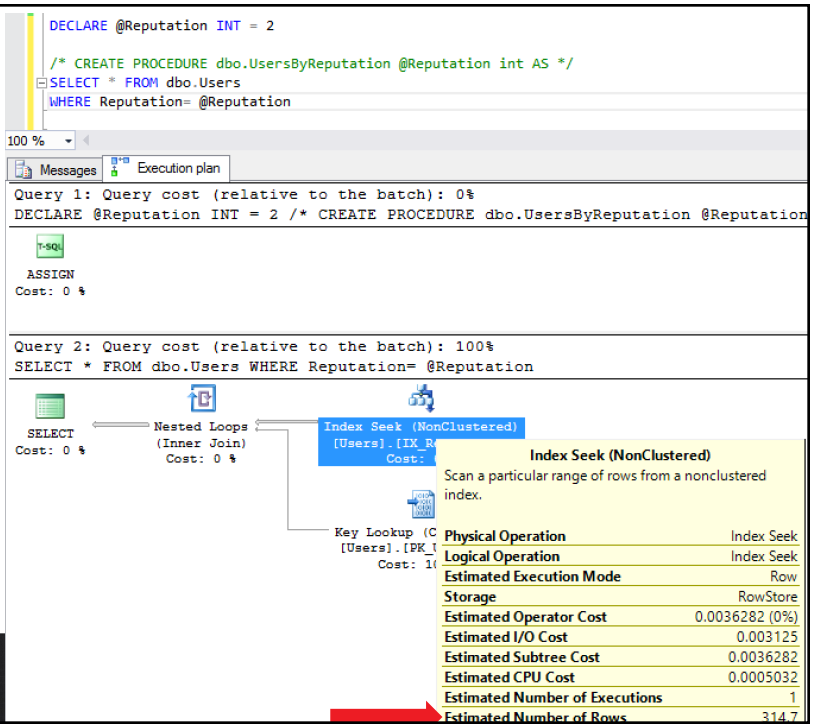
```
DECLARE @Reputation INT = 2

/* CREATE PROCEDURE dbo.UsersByReputation
@Reputation int AS */
SELECT * FROM dbo.Users
WHERE Reputation= @Reputation
```

Check the estimated rows.

It's not for reputation = 1.

It's not for reputation = 2, either.

Where the heck is it getting 314.7 rows?



Local variables use the “density vector”

Rows x Density vector = “unknown” estimate, like an overall average

5,277,831 * 5.962674E-05 = 314.70

```
DBCC SHOW_STATISTICS('dbo.Users', 'IX_Reputation')
```

100 %

Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density
1	IX_Reputation	May 23 2016 4:36PM	5277831	5277831	200	0.04716729

	All density	Average Length	Columns
1	5.962674E-05	4	Reputation
2	1.894718E-07	8	Reputation, Id

That’s not normal

Stored procedures, code with literal values, and parameterized dynamic SQL use the histogram (unless you tell them not to)

But when you use a local variable, it uses the density vector.

It optimizes for an unknown number.

A better way to test: temp stored procs

```
CREATE PROCEDURE #UsersByReputation
  @Reputation INT
AS
SELECT * FROM dbo.Users
WHERE Reputation= @Reputation
GO

EXEC #UsersByReputation 2;
GO
```

EXEC WITH RECOMPILE helps you test different parameter values

```
/* This means:
Don't use an existing plan in cache
Don't cache the plan from this run */

EXEC #UsersByReputation 2 WITH RECOMPILE;
GO


/*Limitation: only applies to the outermost
procedure if you have them nested */
```

4. Options to fix it for good

Option RECOMPILE

```
ALTER PROCEDURE #UsersByReputation
  @Reputation INT
AS
SELECT * FROM dbo.Users
WHERE Reputation= @Reputation
OPTION (RECOMPILE)
GO
```

Option RECOMPILE

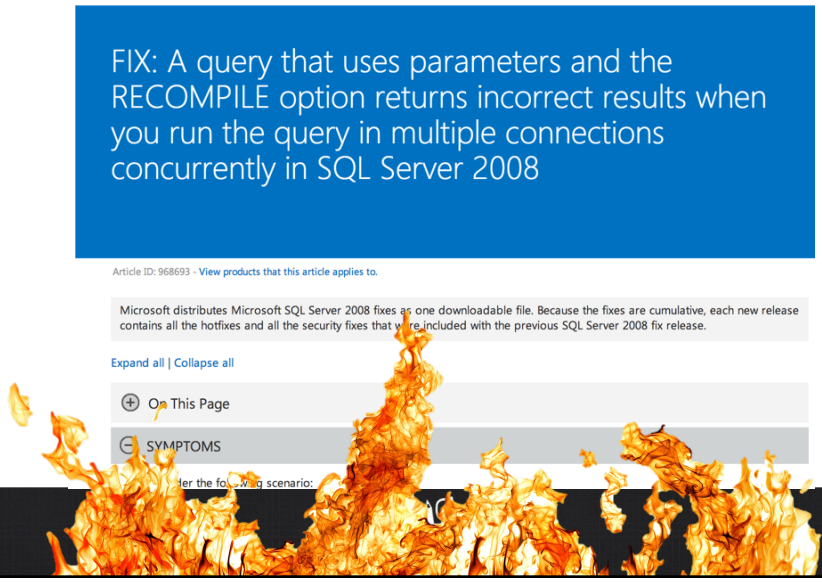
If you MUST use recompile, put it on only on the affected statements (not the stored procedure header)

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)

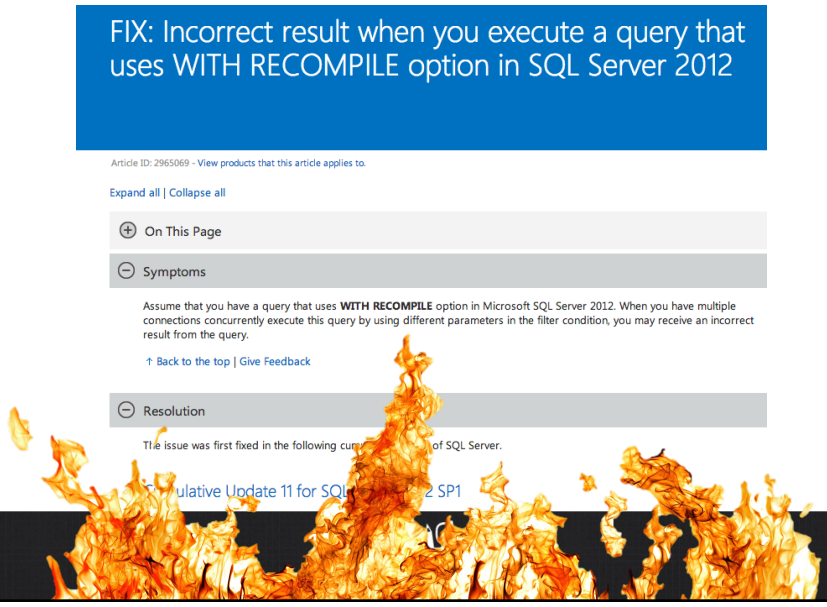
KB 968693 (from 2009)



Whoa.

That'll never happen again, right?

KB 2965069 (from 2014)



OPTIMIZE FOR UNKNOWN

```
ALTER PROCEDURE #UsersByReputation
  @Reputation INT
AS
SELECT * FROM dbo.Users
WHERE Reputation= @Reputation
OPTION (OPTIMIZE FOR UNKNOWN)
GO
```

OPTIMIZE FOR UNKNOWN

This forces everyone to use the density vector (just like a local variable).

In this case, it simply doesn't work:
the density vector gives us a 314 estimate, which isn't anywhere near our real-world numbers for reputation 1 or 2.

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

Fake "OPTIMIZE FOR UNKNOWN"

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

SELECT * FROM dbo.Users
WHERE Reputation= @ReputationUnknown
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

OPTIMIZE FOR Value

```
ALTER PROCEDURE #UsersByReputation
  @Reputation INT
AS
SELECT * FROM dbo.Users
WHERE Reputation= @Reputation
OPTION (OPTIMIZE FOR @Reputation = 1)
```

OPTIMIZE FOR Value

You'd better know your data pretty well.

Your data better not change over time.

Similarly dangerous trick

```
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.)

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 covering index that fits all scenarios.

Recap

You learned 4 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. Don't try to test it with local variables – use a temp stored proc
4. Options for fixing the problem long-term: there's no easy button

Learn more: BrentOzar.com/go/sniff