**[The Elephant and the Mouse, or, Parameter Sniffing in SQL Server](http://www.brentozar.com/archive/2013/06/the-elephant-and-the-mouse-or-parameter-sniffing-in-sql-server/)**

Imagine you work for a zoo, and you need to ship an animal to another zoo. If you are told one will be a mouse and one will be an elephant, you will need to handle them differently. [The Detroit Zoo used a moving van to move two elephants](http://www.thestar.com/news/gta/2012/01/08/detroit_zoo_offers_a_howto_for_shipping_elephants.html), while mice can be put in a box and driven or flown.

**Something Smells Funny**

And it’s not the elephant. SQL Server uses a process called parameter sniffing when it executes stored procedures that have – you guessed it – parameters. When the procedure is compiled or recompiled, the value passed into the parameter is evaluated and used to create an execution plan. That value is then stored with the execution plan in the plan cache. On subsequent executions, that same value – and same plan – is used.

This is a normal, expected behavior in SQL Server. Because compiling queries is expensive, you want plans stored in the cache. You want SQL Server to re-use them as much as possible.

But what happens when the values in a table you’re querying aren’t evenly distributed? What if one value would return 10 rows and another value would return 10,000 rows, or 10 million rows? I call this the elephant and the mouse problem. You would handle one animal differently than the other; SQL Server might create different plans for the queries. But it doesn’t, because you’re using parameters.

[](http://www.flickr.com/photos/paraflyer/386522877/)

What will happen is that the first time the procedure is run and the plan is compiled, whatever value is passed in is stored with the plan. Every time it’s executed, until it’s recompiled, the same value and plan will be used – regardless of whether it is the fastest or best plan for that value. If this is happening to you, and causing performance problems, there are ways to deal with it.

**Parameter Sniffing in Action**

I’m going to run a few queries to show you this behavior. I’ll be using AdventureWorks2012 on a SQL Server 2012 instance.

I’m going to query the Sales.SalesOrderDetail table. I want to know the order numbers and quantity ordered for specific products. My first set of queries will use literal values in the WHERE clause. When using literal values, SQL Server will compile each separately, and store a separate execution plan for each.

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = 897;

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = 945;

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = 870;

1. SELECT SalesOrderDetailID, OrderQty
2. FROM Sales.SalesOrderDetail
3. WHERE ProductID = 897;
4. SELECT SalesOrderDetailID, OrderQty
5. FROM Sales.SalesOrderDetail
6. WHERE ProductID = 945;
7. SELECT SalesOrderDetailID, OrderQty
8. FROM Sales.SalesOrderDetail
9. WHERE ProductID = 870;

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = 897;

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

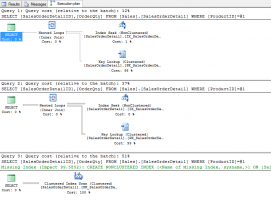
WHERE ProductID = 945;

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = 870;

The same query was run three times. The queries returned 2, 257, and 4,688 rows, respectively. Two different execution plans have been created, even though the query remained the same!



At some point, the optimizer decided it was faster to do a clustered index scan, instead of a nonclustered index seek and a key lookup.

How does this behave when parameterized? I create a stored procedure to test this.

CREATE PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

AS

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID;

1. CREATE PROCEDURE Get\_OrderID\_OrderQty
2. @ProductID INT
3. AS
4. SELECT SalesOrderDetailID, OrderQty
5. FROM Sales.SalesOrderDetail
6. WHERE ProductID = @ProductID;

CREATE PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

AS

SELECT SalesOrderDetailID, OrderQty

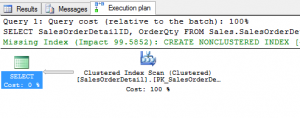
FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID;

I’m going to execute this stored procedure for the first time with the “elephant” – product ID 870, which returns 4,688 rows.

EXEC Get\_OrderID\_OrderQty @ProductID=870EXEC Get\_OrderID\_OrderQty @ProductID=870

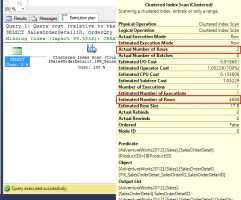
The data is retrieved through a clustered index scan once again.



Now, I’ll execute this stored procedure with the “mouse” – product ID 897, which returns 2 rows.

EXEC Get\_OrderID\_OrderQty @ProductID=897EXEC Get\_OrderID\_OrderQty @ProductID=897

This time, instead of using a nonclustered index seek and a key lookup, the values are retrieved with a clustered index scan. Also note that the estimated number of rows and actual number of rows are very different!



What happened? This is parameter sniffing in action. One value is stored in the execution plan, and that is used to create the plan, regardless of what value is passed in. I can verify this by right-clicking the execution plan and selecting “Show Execution Plan XML”. In the XML, I search for “ParameterCompiledValue”. I find the following line.

Elephant Mouse 4

The compiled value is 870. Until the stored procedure is recompiled, this is the value that will be used.

**Is This Bad?**

The real question is, “Is this bad?” It isn’t – until it is. Depending on the query and the data, each execution of this query may return the results in an acceptable amount of time. It is only when query performance reaches an unacceptable threshold that you may need to do something to prevent it from happening.

**When Parameter Sniffing Stinks**

When parameter sniffing is negatively affecting a stored procedure, what can you do to fix it?

The first option is to do nothing. You can explain the behavior of SQL Server to your end users. They might even listen! But they won’t be happy. They need their data, and they need it faster.

Other options you have are to recompile the stored procedure each time, use a query hint to produce a good enough plan, or do some creative coding. None of these are perfect options – they all have drawbacks. Understand the benefits and drawbacks of each option. Test the variations carefully in a development environment. The last thing you want to do is make the problem worse!

**Recompiling**

You can force SQL Server to recompile the stored procedure each time it is run. The benefit here is that the best query plan will be created each time it is run. However, recompiling is a CPU-intensive operation. This may not be an ideal solution for stored procedures that are run frequently, or on a server that is constrained by CPU resources already. Another thing to remember is that the plans won’t be stored in the cache, which makes them harder to find if they are problematic.

To show this, I’m going to alter my stored procedure to include the WITH RECOMPILE statement.

ALTER PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

WITH RECOMPILE

AS

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID;

1. ALTER PROCEDURE Get\_OrderID\_OrderQty
2. @ProductID INT
3. WITH RECOMPILE
4. AS
5. SELECT SalesOrderDetailID, OrderQty
6. FROM Sales.SalesOrderDetail
7. WHERE ProductID = @ProductID;

ALTER PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

WITH RECOMPILE

AS

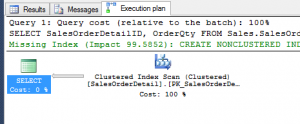
SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID;

What plan will executing this with the “elephant” – product ID 870 – produce?

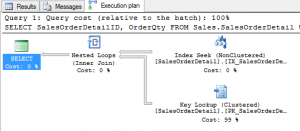
EXEC Get\_OrderID\_OrderQty @ProductID=870EXEC Get\_OrderID\_OrderQty @ProductID=870



A clustered index scan has been performed.

What plan will executing this with the “mouse” – product ID 897 – produce?

exec Get\_OrderID\_OrderQty @ProductID=897exec Get\_OrderID\_OrderQty @ProductID=897



The nonclustered index seek and key lookup are being performed, as expected. This is happening because WITH RECOMPILE tells SQL Server, “Don’t store a plan in cache for me. I’m smart enough to figure this out on my own each time.” Remember, the cost of this is increased CPU usage each time the stored procedure is run.

**Query Hinting**

Another option is to use the OPTIMIZE FOR query hint. This tells SQL Server to use a specified value when compiling the plan. If, through testing, you can find a value that produces a “good enough” plan each time, and the performance is acceptable for both mice and elephants, this is a good option for you.

However, understand that you are bossing the query optimizer around. You are telling it what you think is best. The biggest drawback with OPTIMIZE FOR is on tables where the distribution of data changes. The faster it changes, the more out of date this hint could become. What if the value you provide is not optimal in a month, or a year? You need to have a method in place to regularly review and revise this.

I know that using product ID 945 produces a “good enough” plan for this query, so I alter the procedure to include the OPTIMIZE FOR query hint.

ALTER PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

AS

SELECT SalesOrderDetailID, OrderQty

FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID

OPTION (OPTIMIZE FOR (@ProductID=945));

1. ALTER PROCEDURE Get\_OrderID\_OrderQty
2. @ProductID INT
3. AS
4. SELECT SalesOrderDetailID, OrderQty
5. FROM Sales.SalesOrderDetail
6. WHERE ProductID = @ProductID
7. OPTION (OPTIMIZE FOR (@ProductID=945));

ALTER PROCEDURE Get\_OrderID\_OrderQty

@ProductID INT

AS

SELECT SalesOrderDetailID, OrderQty

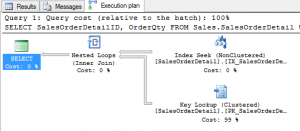
FROM Sales.SalesOrderDetail

WHERE ProductID = @ProductID

OPTION (OPTIMIZE FOR (@ProductID=945));

What plan will the “elephant” – product ID 870 – produce?

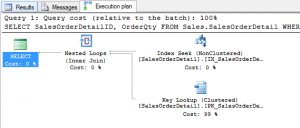
EXEC Get\_OrderID\_OrderQty @ProductID=870EXEC Get\_OrderID\_OrderQty @ProductID=870



A nonclustered index seek and key lookup has been performed. Viewing the XML shows us that the compiled value is 945.

The “mouse” – product ID 897 – has the same execution plan.

EXEC Get\_OrderID\_OrderQty @ProductID=897EXEC Get\_OrderID\_OrderQty @ProductID=897

****

**Be Creative**

Another option may be to create separate stored procedures for each value, or at least for those values you know will produce a specific execution plan. I’m not going to show this. Why? Because it’s ugly, it’s cumbersome, and I don’t recommend it. Who wants to maintain multiple stored procedures, when you can maintain one? What do you do when the distribution of data changes? How do you make the logic clear and easily understood?

**When Parameter Sniffing Stinks**

Understand that parameter sniffing is not a bad thing – it is an integral part of SQL Server. The data in your tables and your queries can lead to parameter sniffing not making sense, however. Understanding how to identify if it is a problem and knowing various ways to fix it when it is a problem are valuable skills to have. When parameter sniffing stinks, test methods to correct it and determine the best way to freshen the air in the server.

1. What is also useful at times is to add the option recompile at the statement level.

The added benefit I believe is that the plan is cached to the point it encounters such a statement and all \_current\_ values of the arguments \_and\_ local variables are taken into account.

Example:

SELECT …  
Table1 Join Table2 ON  
WHERE @Midnight <= col1  
OPTION (recompile)