By Milos Radivojevic

Parameter Sniffing Problem

with SQL Server Stored Procedures

Your users claim that suddenly the execution of one stored procedure is very slow. However, when you execute it within SSMS with the same parameters, it runs quickly. In addition, users said for some parameter combinations the stored procedure performs well. If you are in this scenario, you are discovering the dark side of parameter sniffing.

What is Parameter Sniffing?

Whenever a stored procedure is invoked, the query optimizer tries to reuse an execution plan. If a matching execution plan exists in cache it will be "blindly" reused. If not, a new plan will be generated. During the plan generation the optimizer analyses and optimizes all queries in the stored procedure. It checks possible ways to physically generate a result set and considers several factors: existence of indexes on the columns participating in **JOIN** and **WHERE** clause, table size and data distribution in the involved tables, etc, in order to estimate the query selectivity. When an execution plan for a stored procedure is being generated in addition to all these factors the optimizer considers also parameters that are submitted by the procedure invocation. The optimizer "sniffs" these parameters into the plan considerations. It could be that the parameters don't have any influence on the structure and the selectivity of the resultant query, but usually the parameter values significantly affect the query selectivity and the generated execution plan. Is this bad? It could be. If the parameter values for subsequent calls of the stored procedure have similar values (actually when their values don't change the selectivity of the procedure queries) the parameter sniffing is a feature. Otherwise it could dramatically reduce performance. Let's take the following stored procedure on Adventure Works database as an example:

Listing 1: Creating Sample Stored Procedure

```
CREATE PROCEDURE dbo.getSalesOrderHeader
@SalesOrderID int
AS
            SELECT SalesOrderID,OrderDate,ShipDate,SubTotal
            FROM Sales.SalesOrderHeader
            WHERE SalesOrderID = @SalesOrderID
```

This stored procedure returns four columns from the table **Sales.SalesOrderHeader** for a given **SalesOrderID.** Let's invoke it for the first time with the parameter **45671**. Since the column **SalesOrderID** is a clustered primary key in the table, the execution plan for this procedure is simple: a clustered index seek. See figure 1, page 45.

So, where is there parameter sniffing? If you choose the option *Show Execution Plan XML…* you can get an XML representation of the execution plan. Under the node *ParameterList* you can see that *ParameterCompiledValue* has a value

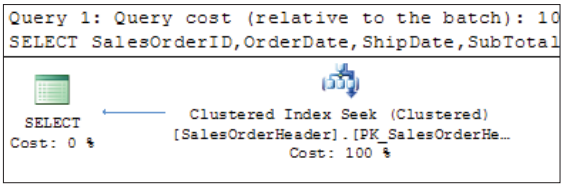

Figure 1: Execution plan for the stored procedure from Listing 1

Listing 2: Creating Stored Procedure With Range Operator

```
CREATE PROCEDURE dbo.getSalesOrderHeader
@OrderDate datetime
AS
          SELECT SalesOrderID,OrderDate,ShipDate,SubTotal
          FROM Sales.SalesOrderHeader
          WHERE OrderDate >= @OrderDate
```

45671 and this parameter was considered by the plan generation.

```
<ParameterList>
  <ColumnReference Column="@SalesOrderID" ParameterCompiledValue="(45671)" ParameterRuntimeValue="(45671)" />
</ParameterList>
```

Figure 2: XML version of the execution plan for the stored procedure from Listing 1 (fragment)

If we call it again with some other parameter (for instance 56781) we can see that *ParameterCompiledValue* still has the value 45671, only *ParameterRuntimeValue* was changed.

```
<ParameterList>
  <ColumnReference Column="@SalesOrderID" ParameterCompiledValue="(45671)" ParameterRuntimeValue="(56781)" />
</ParameterList>
```

Figure 3: XML version of the execution plan for the stored procedure from Listing 1 with a different parameter (fragment)

So, the generated execution plan is optimal for the parameter value 45671. Is this a problem here? Not at all! This plan is also optimal for the value 56781. Actually, for this stored procedure only one execution plan makes sense: a clustered index seek. The procedure returns exactly one row or an empty set, if there are no orders for a given order id. The value of the parameter by the first invocation of the stored procedure doesn't affect the selectivity of the resultant query. That's the reason why in this case the parameter sniffing affect can be ignored. The parameter is of course important to define ***which*** row should be returned but absolutely doesn't affect the way ***how*** to return this row.

However, when different execution plans are possible, then parameter values by the first invocation can decide which one will be used for all subsequent invocations. See listing 2.

If the parameter values for subsequent calls of the stored procedure have similar values (actually when their values don't change the selectivity of the procedure queries) the parameter sniffing is a feature. Otherwise it could dramatically reduce performance.

Now we have two reasonable execution plans:

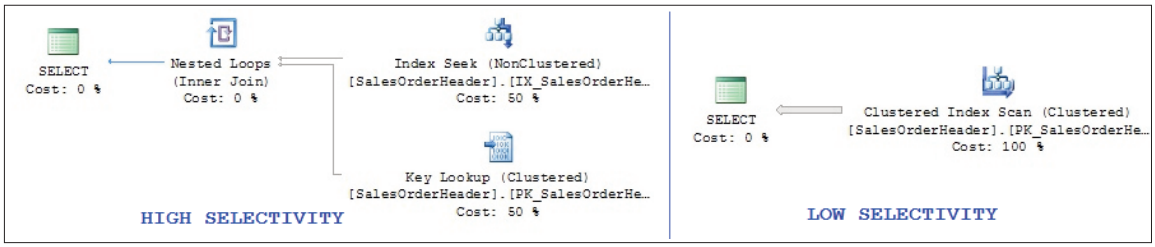

Figure 4: Execution plans for the stored procedure **dbo.getSalesOrderHeader**

Parameter sniffing is not limited to stored procedures only, all parameterized queries can be prone to parameter sniffing: static queries with simple, auto or forced parameterization or dynamic queries called with sp_executesql.

Which one will be used is decided by the value of the parameter **OrderDate** by the first invocation of the stored procedure. If the parameter value is highly selective (few orders in the result set) the second plan will be generated, otherwise the whole table will be scanned (clustered index scan). In this case we cannot ignore the parameter sniffing affect. Some invocations of the stored procedure end up with a non-optimal plan. Usually it is a problem. It could be that the execution plan is optimal for highly selective parameter values and that, let's say, 95% of calls are with the "optimal" parameter value. In this case we can say that these 5% have a problem and this is, from the business logic point of view, not so important for the application. However, the problem is that the execution plan doesn't remain forever in cache. It could be removed from there (for many reasons) and we don't have control over when this happens. So, after the plan is removed from cache, it is a lottery if the next invocation comes with an expected parameter value.

Parameter sniffing is not limited to stored procedures only, all parameterized queries can be prone to parameter sniffing: static queries with simple, auto or forced parameterization or dynamic queries called with **sp_executesql**. In this article we'll describe this phenomenon in stored procedures.

When Parameter Sniffing is a Problem?

There are two types of stored procedures prone to parameter sniffing issues:

- stored procedures with parameters participating in the range operators
- stored procedures with optional parameters

The procedure **dbo.getSalesOrderHeader** belongs to the first group, and we'll describe in this article how to solve the parameter sniffing problem in those stored procedures. In the next SolidQ Journal we'll discuss the parameter sniffing problem in stored procedures with optional parameters.

So, we've already seen that two execution plans are possible for this stored procedure. If the first invocation was with the parameter value

01.07.2001 then all rows are returned and according to this a clustered index scan has been chosen by the optimizer as optimal execution plan. SQL Server had to perform 1.406 logical reads in order to generate a result set. However, when the stored procedure has been invoked for the first time with the value 31.07.2005, an invocation with the parameter 01.07.2001 ends up with 94.456 logical reads! Figure 5 shows the execution time for all combinations of parameter value selectivity and appropriate execution plans:

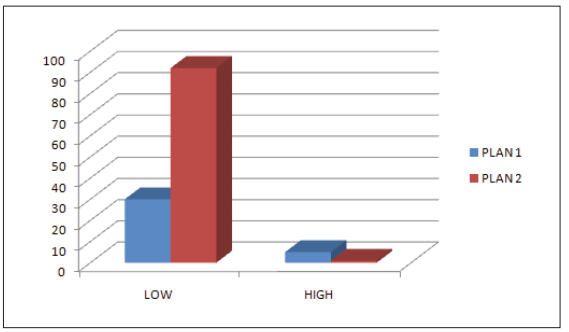

Figure 5: Execution time for low and high selective parameter values

We can see that a call with a low selectivity parameter performs three times better with its optimal plan, while a high selectivity parameter call feels better with an execution plan including an index seek. The parameter sniffing problem means that a stored procedure is executed with a non-optimal plan.

Solving Parameter Sniffing Problem

Before we start solving parameter sniffing problems we have to define ***what*** we want to do. Then we'll see ***how*** to achieve this. One of the misconceptions about parameter sniffing is that the solution for the problem is to disable the effect. This effect is not always bad: it could be very bad, but at the same time it is a feature and disabling solves some problems, but it can degrade performances of well performing queries. Although parameter sniffing is a performance problem, its solution should include business requirements.

As mentioned above the optimizing process :-

- considers parameter values by the first invocation of the stored procedure
- uses a cached plan for all subsequent calls of the stored procedure

This is the default behavior of the optimizer and if we are not satisfied with the product of optimization we have to change one of these factors. We can suggest to the optimizer (or force it) not to consider the input parameters for the plan generation or suggest that it should not use the generated plan for all subsequent calls. In the first case we define some "generic" execution plan or a plan that is generated without knowledge about parameter values and the plan is used for all invocations. Or we can simply force the optimizer to use an optimal execution plan for all parameter combinations.

So, before we start to solve the problem we should define the goal of our optimization i.e. what does solving the problem actually mean. In our stored procedure one solution would be to use a generic execution plan. That means that all calls of the stored procedure will be executed with the execution plan based on the clustered index scan operator. That means that parameter values causing low selectivity will use an optimal plan and the others are chosen as victim and will always use a sub-optimal plan. If this is acceptable from the business logic point of view, than we have defined the goal – using an execution plan optimal for low selectivity.

Approach 1: Affecting Parameter Sniffing Effect and Using One Plan for All Parameter Combinations

By taking this approach we instruct the optimizer to use some specific execution plan (with substituting input parameters) or to ignore input parameters.

Substituting Input Parameters

When we have a popular parameter value combination we can force the optimizer to use this combination whenever it creates an execution plan. In this case we ensure that this combination performs well always, regardless of the existence of the plan in cache. In our case we can decide to optimize execution for lowly selective parameters and to use the plan with the clustered index scan operator (blue plan in figure 5). This means that parameter values causing low selectivity will use an optimal plan and the others are chosen as victims and will always use a suboptimal plan. This is a business logic decision and should be acceptable from the business logic point of view. In this case parameter sniffing is not disabled; instead we substitute input parameter values. The query hint **OPTIMIZE** forces the optimizer to generate an execution plan optimal for the value from the query hint rather than the value submitted by the first invocation.

```
OPTION (OPTIMIZE FOR (@OrderDate='20010101'))
```

With the query hint we substitute the submitted parameter value and parameter sniffing effect is not disabled. The optimizer considers it in the plan generation process; we only change the value of the submitted parameter. This is a solution when we want to ensure that the execution is OK for a favorite parameter combination. This is helpful in scenarios where we identified the most important parameter combination and force an execution plan optimal for this combination, while performance problems with the other combinations are ignored. In Figure 6 we can see that this solution ensures that the plan for low selectivity will be used for all stored procedure calls. See figure 6.

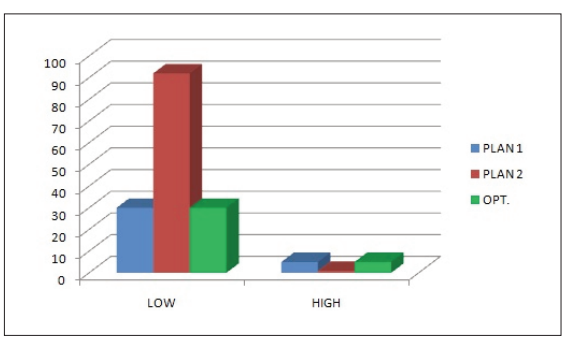

Figure 6: Solution with a generic execution plan

Disable Parameter Sniffing

Disabling of parameter sniffing is helpful when the stored procedure has several parameters and there is not a favorite combination of parameters. If parameter values are not known at the compile time, there is nothing to be sniffed. This is how we exclude them from execution plan considerations. The result of disabling parameter sniffing is, usually, an "average" plan – not optimal, but acceptable for all parameter combinations. We have three options to disable or neutralize parameter sniffing effect: query hints, query rewriting and trace flag 4136.

With Query Hint

In order to disable parameter sniffing we can again use the **OPTIMIZE** query hint.

```
OPTION (OPTIMIZE FOR UNKNOWN)
```

This time the optimizer receives a suggestion to ignore submitted parameter values. Therefore the value is unknown and the optimizer uses the density of table values to estimate cardinality. For the range operator that means about 30% of rows are expected in the result set which usually means an average execution plan with clustered index scan. This option is available in SQL Server 2008.

Query Rewriting

The same effect can be achieved by rewriting some code in the stored procedure. See listing 3, page 49.

Listing 3: Modifying Stored Procedure By Using Local Variables

```
CREATE PROCEDURE dbo.getSalesOrderHeader
@OrderDate datetime
AS
          DECLARE @OrderDateLocal AS datetime = @OrderDate
          SELECT SalesOrderID,OrderDate,ShipDate,SubTotal
          FROM Sales.SalesOrderHeader
          WHERE OrderDate >= @OrderDateLocal
```

We introduce a local variable and assign the value of the procedure parameter to it. Since the optimizer compiles the stored procedure at the batch level, the value of the argument in the **WHERE** clause is not known. Therefore we get the same plan as with the **UNKNOWN** hint.

Trace Flag 4136

In October 2010, as part of the cumulative update Microsoft provided an option to disable parameter sniffing at the instance level (KB980653). When trace flag 4136 is on, parameter sniffing effect is disabled for the whole instance. That has the same effect as putting the **OPTIMIZE FOR UNKNOWN** hint in all stored procedures and queries submitted to the database engine. This sounds very restrictive and dangerous and therefore there are still exceptions where this trace flag has no effect: queries using the **OPTIMIZE FOR** or **RECOMPILE** query hints and stored procedures defined with the **WITH RECOMPILE** option. Although this brings necessary flexibility and does not break up our already implemented optimizing efforts, the trace flag 4136 affects the whole server instance and should be considered as a last resort for solving parameter sniffing problems.

The approach with query hints has advantages over the option with local variables because query hints don't affect business logic in a stored procedure and can be implemented outside of the stored procedure. By combining the query hint with another great SQL Server feature – plan guides - we can solve parameter sniffing problems without changing application code. This is very important for systems where code changes are not allowed or where a small code change is unacceptably expensive (project-recompile, testing, deployment etc.).

Approach 2: An Optimal Plan for Every Parameter Combination

When it is required that every combination should have an optimal plan, disabling parameter sniffing cannot facilitate this. We have two options to implement this: instruct the optimizer to generate (recompile) a new plan for every stored procedure call or rewrite the stored procedure by using another stored procedure (decision tree stored procedure).

By combining the query hint with another great SQL Server feature – plan guides - we can solve parameter sniffing problems without changing application code.

Recompile of Execution Plan

This can be done with the query hint **OPTION (RECOMPILE)** or by including the option **WITH RECOMPILE** in the definition of the stored procedure.

Listing 4: Modifying Stored Procedure by Changing Stored Procedure Definition

```
CREATE PROCEDURE dbo.getSalesOrderHeader
@OrderDate datetime
WITH RECOMPILE
AS
          SELECT SalesOrderID,OrderDate,ShipDate,SubTotal
          FROM Sales.SalesOrderHeader WHERE OrderDate >= @OrderDate
```

Listing 5: Modifying Stored Procedure by Using Query Hint

```
CREATE PROCEDURE dbo.getSalesOrderHeader
@OrderDate datetime
AS
          SELECT SalesOrderID,OrderDate,ShipDate,SubTotal
          FROM Sales.SalesOrderHeader WHERE OrderDate >= @OrderDate
          OPTION (RECOMPILE)
```

In both cases a new plan is generated for every procedure invocation. The difference is that **OPTION (RECOMPILE)** is performed at the statement level and will not regenerate all statements in a stored procedure but only the one marked with this option. In our case it is the same, we have only one statement. In the next SolidQ Journal we'll see advantage of compiling at the statement level. Figure 7 shows that the recompile-plan combines the initial plans for low and high selective parameters and is optimal for all parameters.

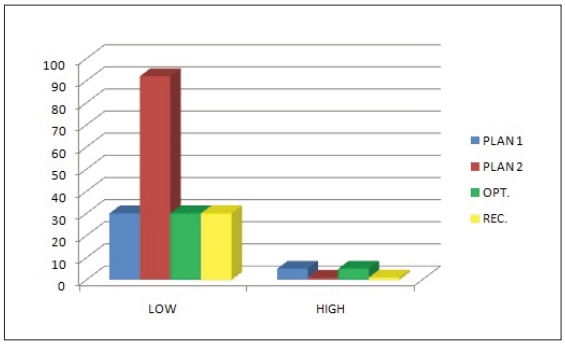

Figure 7: Solution with an optimal plan for each parameter value combination

Decision Tree Stored Procedure

A decision tree stored procedure decides which sub-procedure should be called based on submitted parameters. Let's show this solution. See listing 6, page 51.

The procedure determines if the submitted date is more than 10 days in the past. If so, it calls a stored procedure for low selectivity parameters, otherwise the version for high selective dates is selected. The sub-procedures have the same procedure body and are identical to the initial stored procedure **dbo.getSalesOrderHeader**. The Figure 8 shows the execution plans for high and low selective parameters.

Listing 6: Modifying Stored Procedure by Using a Decision Tree Stored Procedure

```
CREATE PROCEDURE dbo.getSalesOrderHeaderDT
@OrderDate datetime
AS
        IF @OrderDate > DATEADD(day,-10,CURRENT_TIMESTAMP)
                EXEC dbo.getSalesOrderHeaderHighSelectivity @OrderDate
        ELSE
                EXEC dbo.getSalesOrderHeaderLowSelectivity @OrderDate
```

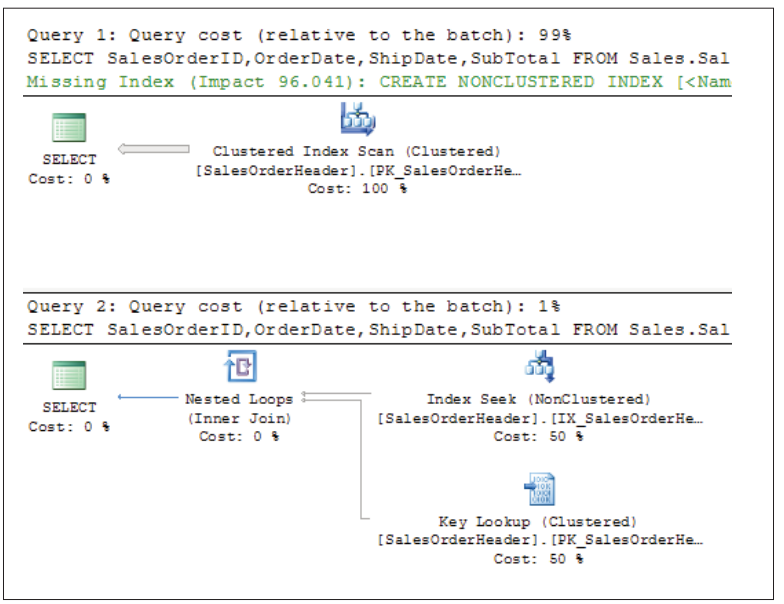

Figure 8: Execution time for low and high selective parameter values for the decision tree stored procedure

This solution allows reusing of execution plans and from a performance point of view it's better than the version using recompile option. However, this approach is open to maintenance problems. For every combination where we want to reuse an existing plan we would need one sub-procedure. Therefore, the solution would be unmanageable. Another maintenance problem is that decisions about what is highly selective are not so easy and they should be evaluated and regularly checked to see if they are still appropriate. We can combine both approaches by creating enough sub-procedures to cover most common plans and one with recompile option for all other combinations. Still, the number of procedures to cover the most common plans may become quickly unmanageable.

We've seen in this article what is parameter sniffing and when and why it is a problem. We have discussed different solutions for parameter sniffing problems with stored procedures using range operators. Next month we'll see why stored procedures with optional parameters are prone to the parameter sniffing problem and we'll again offer several solutions for this problem. ■

About the Author

Milos Radivojevic *is a Data Platform Architect with SolidQ CEE, located in Vienna, Austria. His primary focus is on database development and performance tuning for OLTP systems.*