Fixing Cache Bloat Problems With Guide Plans and Forced Parameterization

20 February 2014
by Fabiano Amorim

Imagine it. You've been asked to fix a dire performance problem with a SQL Server database. You find a severe case of 'Cache Bloat' due to ad-hoc queries, but you can't fix the code itself. What should you do? Specify forced parameterization? Perhaps a better idea would be to use guide plans.

'Cache bloat' is a problem that I've found in the vast majority of those SQL Server instances that I have been asked to work on in order to tune their performance. Despite all the recommendations published on the web that warn people to be careful about this, it is very common to find that developers still run ad-hoc queries in such a way that they bloat the cache with lots of query plans. It is frustrating to see because it is so easily avoided. Sometimes, however, you're not in a position where you can change the code.

Why is it important to avoid Cache Bloat? How can the DBA avoid the consequences of this problem without requiring the developer to change the application? We'll answer these questions in this article.

So, what is cache bloat ?

'Cache bloat' is a term used to describe a growth in the size of SQL Server's procedure cache, using memory 'stolen' from the buffer cache. There is a memory area that is used to store execution plans that are created by SQL Server. It is also used for a variety of other volatile storage such as locks, connections and cached data. SQL Server will take space from this shared buffer pool for the extra plans that need to be stored, and at a certain point this will start to restrict the memory available for other processes. This could cause a very high CPU usage without any blocking, slow performance, and finally errors in executing queries.

The procedure cache is used by the relational engine to avoid having to compile an execution plan every time a particular query is executed. Before a plan is created for a query, the cache is searched to see if there is already a plan for the same query. If so, that plan is reused.

Sometimes this plan cache area of memory becomes filled with queries that the query optimizer doesn't reckon to be the same, but which seem obviously the same to the programmer. To reuse a plan, a query need to be equal to what's in the cache: for instance, if a whitespace or an upper/lower letter is different it will not reuse the plan, and it will trigger a plan compilation. Although the algorithm for determining whether the queries are the same is very efficient, certain things, such as unqualified object references, will lead it to consider them different.

The most obvious 'surface' difference between two queries that actually disguises the fact that they are the same is the 'literal' or value in a SQL query. So the query..

```
SELECT * FROM Sales.SalesOrderHeader WHERE OrderDate ='12 Mar 2004'
```

… should really be considered the same as..

```
SELECT * FROM Sales.SalesOrderHeader WHERE OrderDate ='11 Apr 2004'
```

The literal date '12 Mar 2004' is different but the query plan should be the same. The Relational engine uses a technique called 'auto-parameterization' to turn the literal value into a parameter so that subsequent calls of the same SQL with a different date literal will be matched, but it only does so if it judges that they should be handled by the same query plan.

Normally, if you are using procedures, functions or parameterized queries/batches, you're providing a strong hint to the query engine about where the parameters are and whether it is safe to reuse a plan. Ad-hoc queries don't provide many clues, and SQL Server likes to play safe and assume that each one requires a new plan. Unless SQL Server determines that it can automatically parameterize a query, or it determines that it is the same query, it is forced to generate a new execution plan. Apart from wasting resources to compile the query (which consumes a lot of CPU), it will store each new plan in cache. If a server receives a large number of ad-hoc queries, it will take pages of memory from the buffer cache pool to store them and this can lead to a more general memory stress.

What's the problem with ad-hoc queries?

Here is a sample of an ad-hoc query:

```
DECLARE @Var VarChar(10), @SQL VarChar(200)
SET @Var = CONVERT(VarChar(10), GETDATE() - (CheckSUM(NEWID()) / 1000000), 112)

SET @SQL = 'SELECT * FROM OrdersBig WHERE OrderDate = ' + '''' + @Var + ''''
EXEC (@SQL)
GO
```

You'll see that the user is mistakenly concatenating the user input to the SQL code 'on the fly' to create the query. If you are forced to use dynamic sql for some reason, sp_executeSQL can be used, with parameters , to allow plan reuse and minimise SQL Injection risks. Even better would be a stored procedure, whose execution plan only needs to be created once, if a plan for it is not found in the procedure cache. Every time it is called, the relational engine reuses the same query plan.

In order to show how Ad-hoc queries can be a problem, I'll use SQL Query Stress to simulate many users running a query at the same time and let's see what happens with CPU and memory. The query uses the SQL Server demo database AdventureWorks2012, and is as follows:

```
SELECT SalesOrderHeader.SalesPersonID,
       COUNT(DISTINCT SalesOrderHeader.CustomerID),
       SUM(SalesOrderDetail.OrderQty)
  FROM Sales.SalesOrderHeader
 INNER JOIN Sales.SalesOrderDetail
    ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
 INNER JOIN Production.Product
    ON Product.ProductID = SalesOrderDetail.ProductID
 WHERE Product.Name = 'C1DCB640-A394-4C33-95B0-D8EFF1DE6895'
 GROUP BY SalesOrderHeader.SalesPersonID
```

The result and logic of the query don´t matter here, I only want to show a query that receives a parameter filter dynamically, to do this lets run it using EXEC and concatenating the filter on Product.Name. Here is the dynamic query:

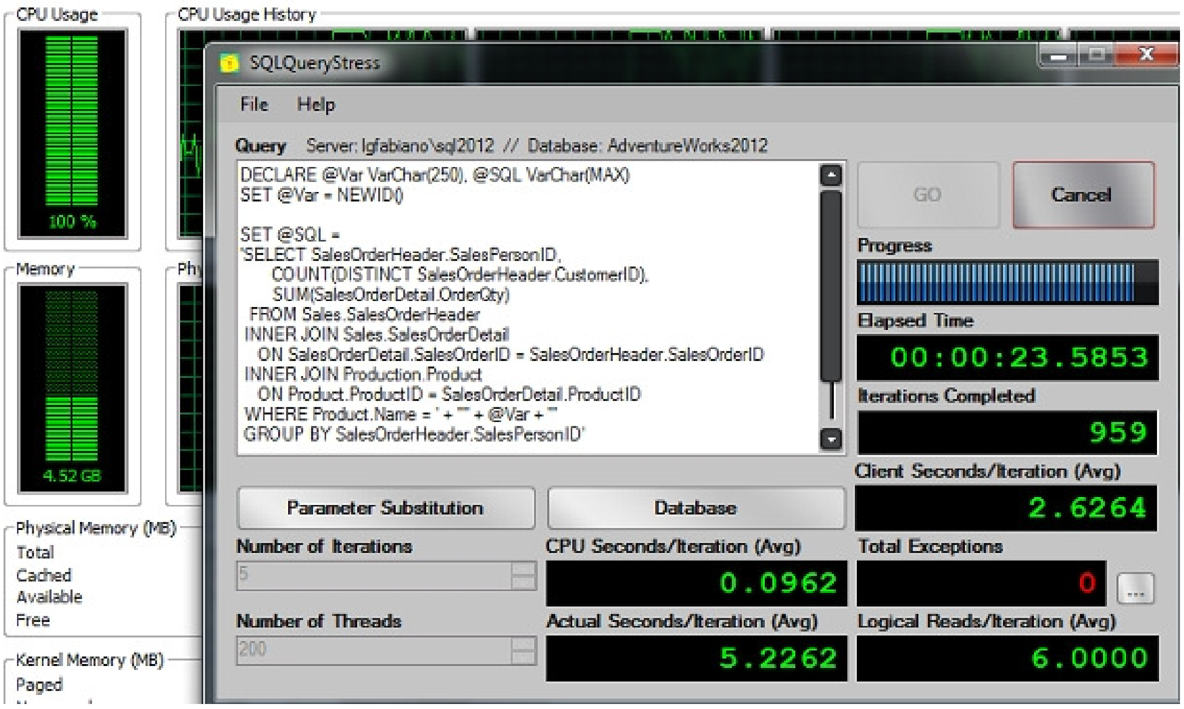
```
DECLARE @Var VarChar(250), @SQL VarChar(MAX)
SET @Var = NEWID()

SET @SQL =
'SELECT SalesOrderHeader.SalesPersonID,
       COUNT(DISTINCT SalesOrderHeader.CustomerID),
       SUM(SalesOrderDetail.OrderQty)
  FROM Sales.SalesOrderHeader
 INNER JOIN Sales.SalesOrderDetail
    ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
 INNER JOIN Production.Product
    ON Product.ProductID = SalesOrderDetail.ProductID
 WHERE Product.Name = ' + '''' + @Var + '''
 GROUP BY SalesOrderHeader.SalesPersonID'

EXEC (@SQL)
```

Every time the query above is executed, SQL Server will run a different query (because of the NEWID that creates a GUID), and consequently will compile a new plan for the query.

If I run the code on **SQLQueryStress**using 5 iterations on 200 threads looks what happens:



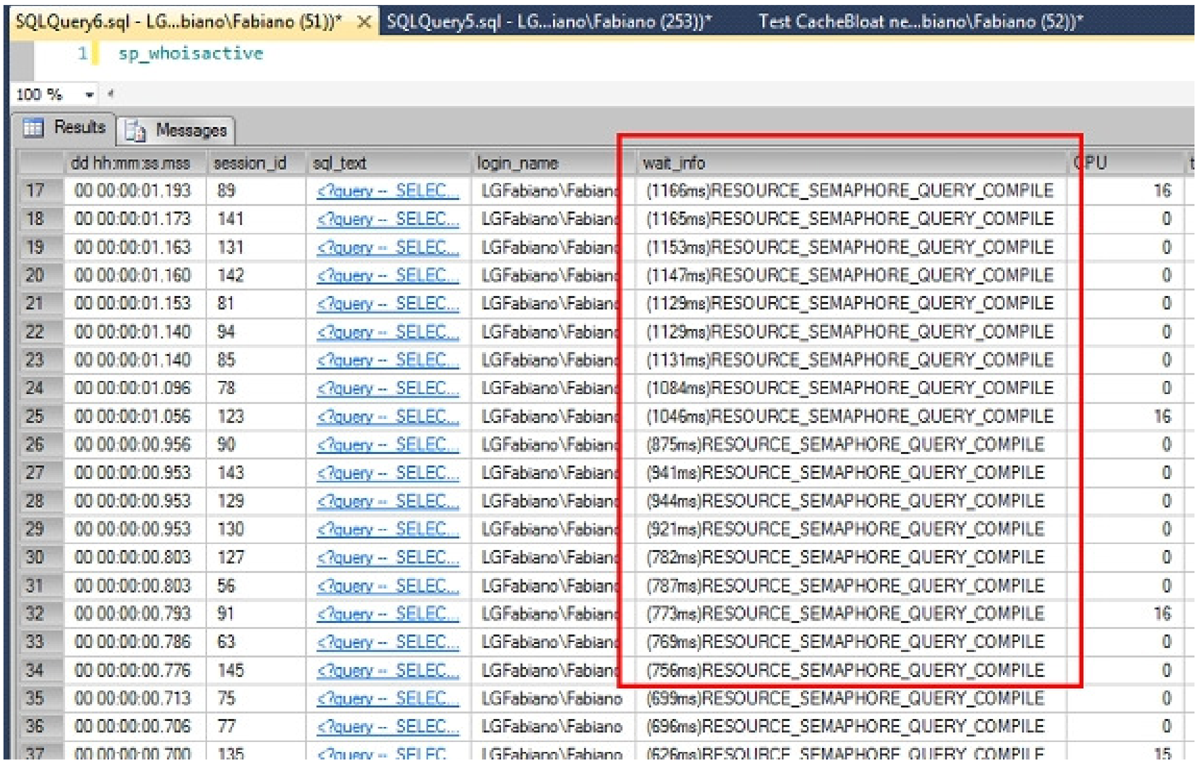
Query   Server: lgfabiano\sql2012 // Database: AdventureWorks2012

```
DECLARE @Var VarChar(250), @SQL VarChar(MAX)
SET @Var = NEWID()

SET @SQL =
'SELECT SalesOrderHeader.SalesPersonID,
       COUNT(DISTINCT SalesOrderHeader.CustomerID),
       SUM(SalesOrderDetail.OrderQty)
  FROM Sales.SalesOrderHeader
 INNER JOIN Sales.SalesOrderDetail
    ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
 INNER JOIN Production.Product
    ON Product.ProductID = SalesOrderDetail.ProductID
 WHERE Product.Name = ' + '''' + @Var + ''''
 GROUP BY SalesOrderHeader.SalesPersonID'
```

Progress
Elapsed Time: 00:00:23.5853
Iterations Completed: 959
Client Seconds/Iteration (Avg): 2.6264
Number of Iterations: 5
CPU Seconds/Iteration (Avg): 0.0962
Total Exceptions: 0
Number of Threads: 200
Actual Seconds/Iteration (Avg): 5.2262
Logical Reads/Iteration (Avg): 6.0000

Since SQL Server has to create an execution plan for each query, it is taking lot of time (more than 23 seconds) and resource (note CPU on 100%) to run the queries. In addition, if you look at the waits for the queries we can see that it is waiting "for the query to compile" (wait **`resource_semaphore_query_compile`**), following is the result of `sp_whoisactive` when the queries were running.


SQLQuery6.sql - LG...biano\Fabiano (51))*
SQLQuery5.sql - LG...iano\Fabiano (253))*
Test CacheBloat ne...biano\Fabiano (52))*
100 %
Results
Messages

```
1   sp_whoisactive
```

	dd hh:mm:ss.mss	session_id	sql_text	login_name	wait_info	CPU
17	00 00:00:01.193	89	<?query -- SELEC...	LGFabiano\Fabian	(1166ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	16
18	00 00:00:01.173	141	<?query -- SELEC...	LGFabiano\Fabian	(1165ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
19	00 00:00:01.163	131	<?query -- SELEC...	LGFabiano\Fabian	(1153ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
20	00 00:00:01.160	142	<?query -- SELEC...	LGFabiano\Fabian	(1147ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
21	00 00:00:01.153	81	<?query -- SELEC...	LGFabiano\Fabian	(1129ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
22	00 00:00:01.140	94	<?query -- SELEC...	LGFabiano\Fabian	(1129ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
23	00 00:00:01.140	85	<?query -- SELEC...	LGFabiano\Fabian	(1131ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
24	00 00:00:01.096	78	<?query -- SELEC...	LGFabiano\Fabian	(1084ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
25	00 00:00:01.056	123	<?query -- SELEC...	LGFabiano\Fabian	(1046ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	16
26	00 00:00:00.956	90	<?query -- SELEC...	LGFabiano\Fabian	(875ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
27	00 00:00:00.953	143	<?query -- SELEC...	LGFabiano\Fabian	(941ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
28	00 00:00:00.953	129	<?query -- SELEC...	LGFabiano\Fabian	(944ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
29	00 00:00:00.953	130	<?query -- SELEC...	LGFabiano\Fabian	(921ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
30	00 00:00:00.803	127	<?query -- SELEC...	LGFabiano\Fabian	(782ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
31	00 00:00:00.803	56	<?query -- SELEC...	LGFabiano\Fabian	(787ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
32	00 00:00:00.793	91	<?query -- SELEC...	LGFabiano\Fabian	(773ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	16
33	00 00:00:00.786	63	<?query -- SELEC...	LGFabiano\Fabian	(769ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
34	00 00:00:00.776	145	<?query -- SELEC...	LGFabiano\Fabian	(756ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
35	00 00:00:00.713	75	<?query -- SELEC...	LGFabiano\Fabiano	(699ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
36	00 00:00:00.706	77	<?query -- SELEC...	LGFabiano\Fabiano	(696ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	0
37	00 00:00:00.700	135	<?query -- SELEC...	LGFabiano\Fabiano	(626ms)RESOURCE_SEMAPHORE_QUERY_COMPILE	15

The main points here are that you're going to have to avoid

- compiling the query every time you execute it since they should be considered to be the same. Because you are only changing the filter predicate, the existing plan is still usable
- wasting CPU on compilation
- the wait for the queries to compile
- wasting memory used to store 1k "different" plans on plan cache

So how do you solve the problem without changing the code?

There are some solutions for this problem, “optimizing for adhoc workloads”, changing the database parameterization to ‘forced’, or just putting the query in either a stored procedure or parameterized batch, and receiving the predicate filter in an input parameter.

The solution I would like to present you is to use guide plans to force the parameterization only for specific queries. The advantage of this technique is that it can be used in cases where you don’t have the option of fixing the code, as with third-party software. You can, of course, set parameterization to ‘forced’ at database level to solve this problem, but it is a blunt instrument to fix a problem that may be due to just a handful of queries. It will force the parameterization for all queries running on the database, whether they are suitable or not, and this may lead in turn to other problems; see a sample here from PFE Thomas Stringer. If you specify forced parameterization you may force plan reuse even where it would result in a poorly-performing plan being selected. You’d be fixing one problem but making others (with other queries).That is why I like to fix the problem using guide plans. With guide plans you can specify the forced parameterization within the scope of individual query

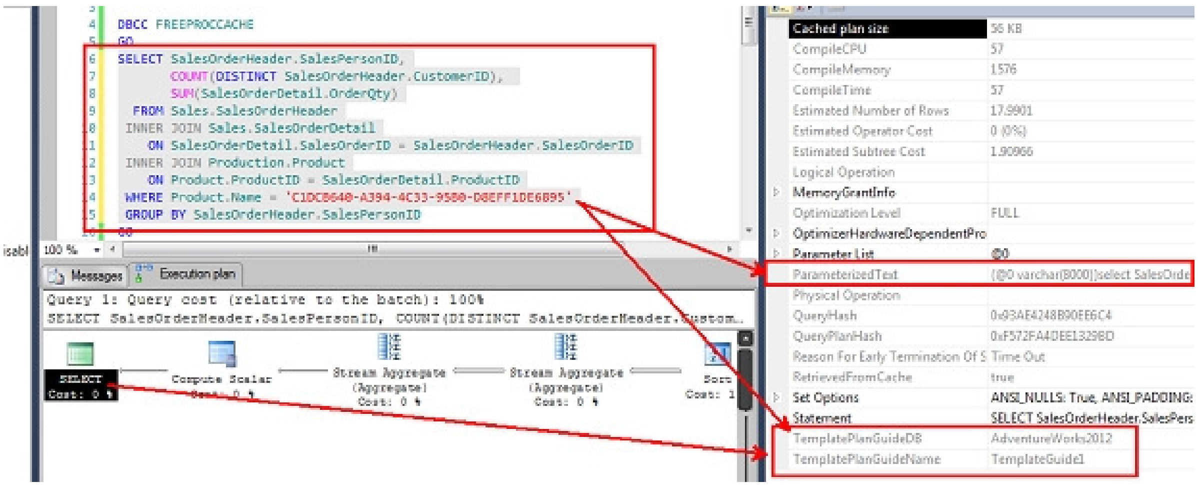
The creation of a template guide plan is straightforward, and to illustrate this, here is a script to create a guide plans to the query used on our tests:

```
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
N'SELECT SalesOrderHeader.SalesPersonID,
       COUNT(DISTINCT SalesOrderHeader.CustomerID),
       SUM(SalesOrderDetail.OrderQty)
  FROM Sales.SalesOrderHeader
 INNER JOIN Sales.SalesOrderDetail
    ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
 INNER JOIN Production.Product
    ON Product.ProductID = SalesOrderDetail.ProductID
 WHERE Product.Name = ''3514C79D-12C2-4673-A5E3-8961F392B396''
 GROUP BY SalesOrderHeader.SalesPersonID',
 @stmt OUTPUT, @params OUTPUT;

EXEC sp_create_plan_guide
    N'TemplateGuide1',
    @stmt,
    N'TEMPLATE',
    NULL,
    @params,
    N'OPTION(PARAMETERIZATION FORCED)';
GO
```

First I’m using the procedure **`sp_get_query_template`** to get a parameterized version of the query and then I’m using the query (variable **`@stmt`**) and parameters (variable **`@params`**) to create the guide plan using the procedure **`sp_create_plan_guide`**.

The result is that, next time you run the query independent of the predicate value specified SQL Server will identify the guide plan and use the hint “parameterization forced” to parameterize the query. You can see that it is using the guide plan and the parameters on the estimated graph execution plan, for instance:

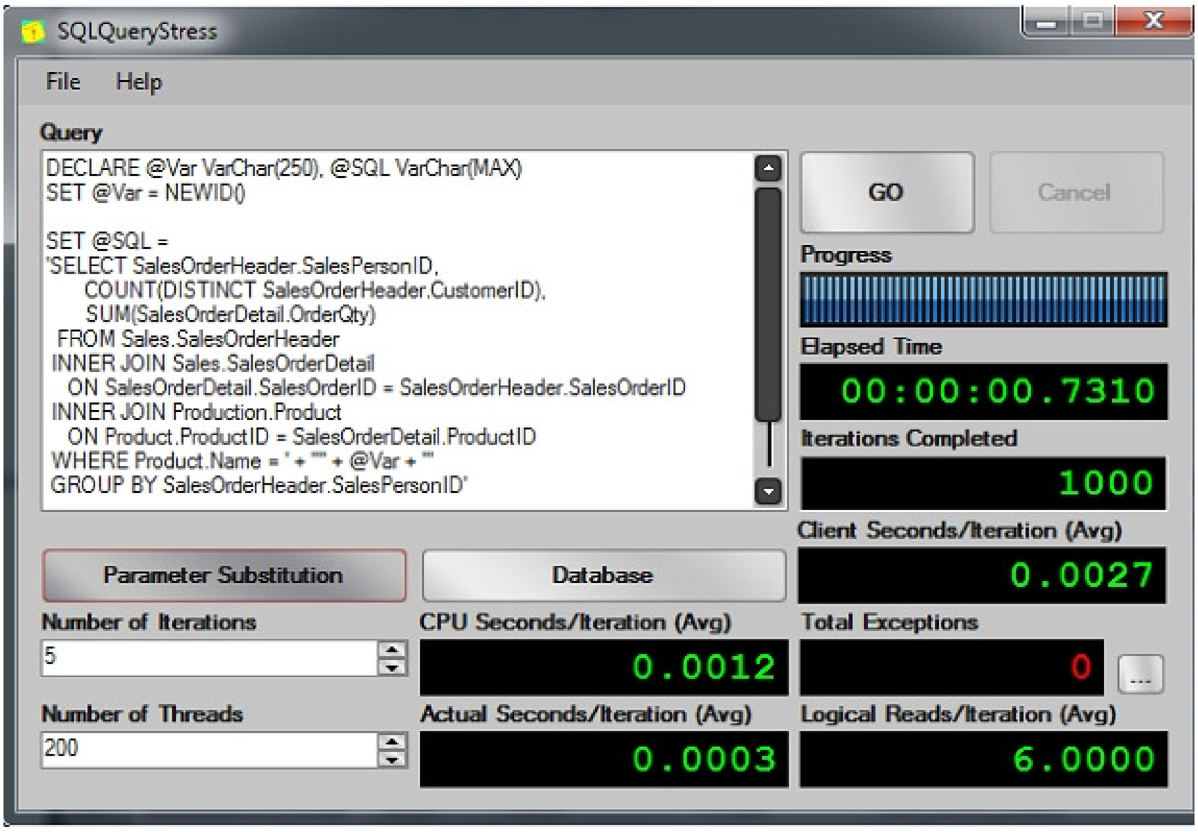


```
DBCC FREEPROCCACHE
GO
SELECT SalesOrderHeader.SalesPersonID,
       COUNT(DISTINCT SalesOrderHeader.CustomerID),
       SUM(SalesOrderDetail.OrderQty)
  FROM Sales.SalesOrderHeader
 INNER JOIN Sales.SalesOrderDetail
    ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
 INNER JOIN Production.Product
    ON Product.ProductID = SalesOrderDetail.ProductID
 WHERE Product.Name = 'C1DC8640-A394-4C33-9580-D8EFF1DE6895'
 GROUP BY SalesOrderHeader.SalesPersonID
GO
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT SalesOrderHeader.SalesPersonID, COUNT(DISTINCT SalesOrderHeader.Custom...
```

Cached plan size	56 KB
CompileCPU	57
CompileMemory	1576
CompileTime	57
Estimated Number of Rows	17.9901
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	1.90966
Logical Operation	
MemoryGrantInfo	
Optimization Level	FULL
OptimizerHardwareDependentPro	
Parameter List	@0
ParameterizedText	(@0 varchar(8000))select SalesOrde
Physical Operation	
QueryHash	0x93AE4248B90EE6C4
QueryPlanHash	0xF572FA40EE13298D
Reason For Early Termination Of S	Time Out
RetrievedFromCache	true
Set Options	ANSI_NULLS: True, ANSI_PADDING:
Statement	SELECT SalesOrderHeader.SalesPers
TemplatePlanGuideDB	AdventureWorks2012
TemplatePlanGuideName	TemplateGuide1

Now, if I run the same test I did on **SQLQueryStress**, look at the result:



Query

```
DECLARE @Var VarChar(250), @SQL VarChar(MAX)
SET @Var = NEWID()

SET @SQL =
'SELECT SalesOrderHeader.SalesPersonID,
     COUNT(DISTINCT SalesOrderHeader.CustomerID),
     SUM(SalesOrderDetail.OrderQty)
 FROM Sales.SalesOrderHeader
INNER JOIN Sales.SalesOrderDetail
   ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
INNER JOIN Production.Product
   ON Product.ProductID = SalesOrderDetail.ProductID
WHERE Product.Name = ' + '''' + @Var + ''''
GROUP BY SalesOrderHeader.SalesPersonID'
```

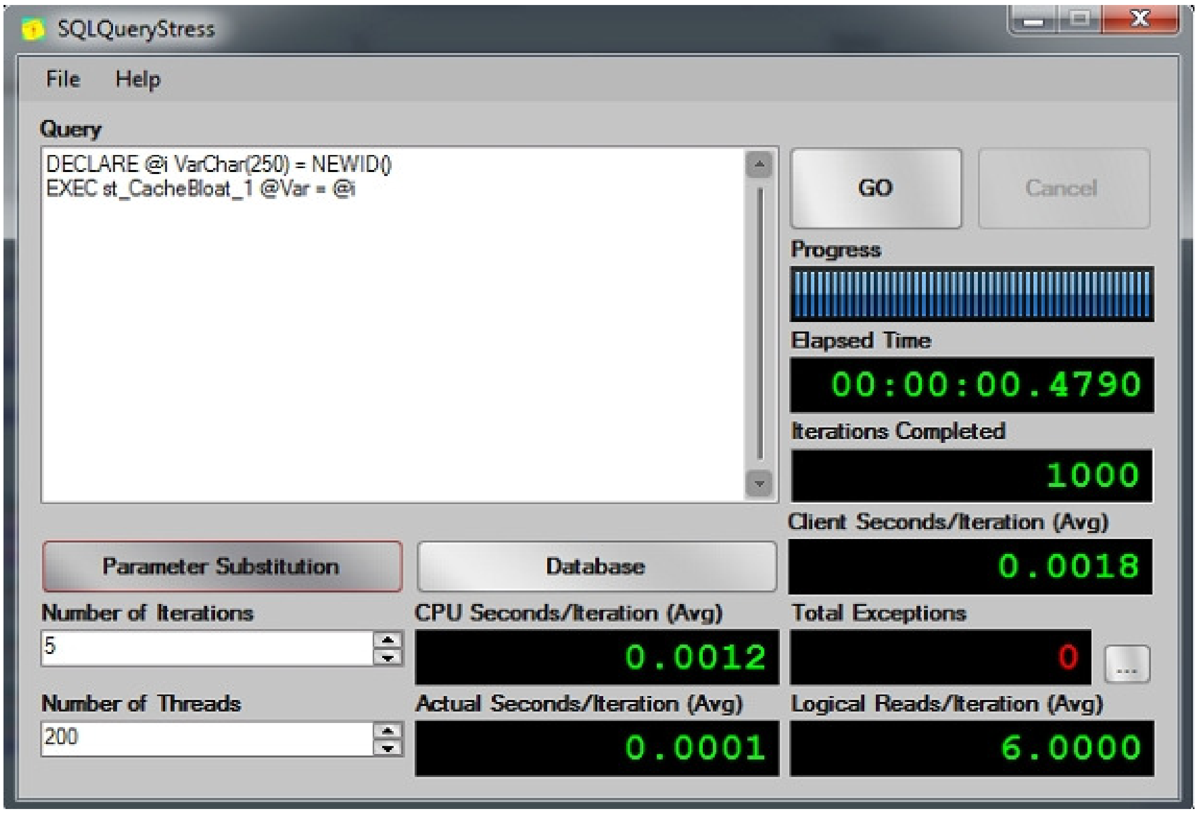
Elapsed Time: 00:00:00.7310
Iterations Completed: 1000
Client Seconds/Iteration (Avg): 0.0027
Number of Iterations: 5
CPU Seconds/Iteration (Avg): 0.0012
Total Exceptions: 0
Number of Threads: 200
Actual Seconds/Iteration (Avg): 0.0003
Logical Reads/Iteration (Avg): 6.0000

All 1000 executions finish on less than a second.

In case you are wondering about the stored procedures and how it would perform on this query, here is the test. First let´s create the procedure:

```
IF OBJECT_ID('st_CacheBloat_1') IS NOT NULL
  DROP PROCEDURE st_CacheBloat_1
GO
CREATE PROCEDURE st_CacheBloat_1 @Var VarChar(250)
AS
BEGIN
  SELECT SalesOrderHeader.SalesPersonID,
         COUNT(DISTINCT SalesOrderHeader.CustomerID),
         SUM(SalesOrderDetail.OrderQty)
    FROM Sales.SalesOrderHeader
   INNER JOIN Sales.SalesOrderDetail
      ON SalesOrderDetail.SalesOrderID = SalesOrderHeader.SalesOrderID
   INNER JOIN Production.Product
      ON Product.ProductID = SalesOrderDetail.ProductID
   WHERE Product.Name = @Var
   GROUP BY SalesOrderHeader.SalesPersonID
END
```

And then executing on SQL Query Stress:



Query
```
DECLARE @i VarChar(250) = NEWID()
EXEC st_CacheBloat_1 @Var = @i
```

Elapsed Time: 00:00:00.4790
Iterations Completed: 1000
Client Seconds/Iteration (Avg): 0.0018
Number of Iterations: 5
CPU Seconds/Iteration (Avg): 0.0012
Total Exceptions: 0
Number of Threads: 200
Actual Seconds/Iteration (Avg): 0.0001
Logical Reads/Iteration (Avg): 6.0000

Again, less than a second. if you are not using Stored procedures, or parameterized queries, then think again.

Conclusion

In general, ad-hoc queries are bad for performance and you should instead use stored procedures or parameterized queries/batches. This is because, to get performance from any database, you should give the relational engine all the help you can. It needs to save the time and resources spent in creating query plans, by only doing it when it's necessary. There is no magical way it can detect that repeated queries are actually the same barring a change in parameter, and that the parameter is not going to change sufficiently to render the existing plan a poor one. You need to tell it. The problem is that sometimes you can't change the code and in this case guide plans, ad-hoc workloads parameter or even forced parameterization is a good alternative.

It is important to understand that may cause other problems related to the parameter sniffing problem, but it is a topic for another article and this is issue already well covered in articles on the internet.

That´s all folks.