**[A Play Around With Table Variable Deferred Compilation in SQL 2019](http://www.sqlservercentral.com/blogs/sql-undercover/2018/10/16/a-play-around-with-table-variable-deferred-compilation-in-sql-2019/)**

* [david.fowler 42596](http://www.sqlservercentral.com/blogs/sql-undercover/)
* Posted on 16 October 2018

FROM: <http://www.sqlservercentral.com/blogs/sql-undercover/2018/10/16/a-play-around-with-table-variable-deferred-compilation-in-sql-2019/?utm_source=SSC&utm_medium=pubemail>



Table variables, they’re nasty, dirty little things that perform about as well as a Robin Reliant in a crosswind, right?  Well you’re not going to have me argue with you there.

But how many of us have got these little beasts floating around in their production code with absolutely nothing that we can do about them, my hand’s in the air on that one.

**So What’s The Issue With Table Variables?**

It’s been well documented so I don’t really want to go too deeply into the issue but basically table variables don’t have statistics.  That really upsets the query optimizer and because it doesn’t know any better, will just assume that the table variable only has a single row.  This can cause all sorts of issues when it works out the query plan.

Let’s have a little look at what happens when we run the following code against our test database.  We’re using Kendra Little’s neat [babbynames database](https://littlekendra.com/2017/02/28/the-babbynames-sample-database-now-on-github/) to test on in compatibility level 140.

DECLARE @t1 TABLE (FirstNameId INT, FirstName VARCHAR(255))

INSERT INTO @t1

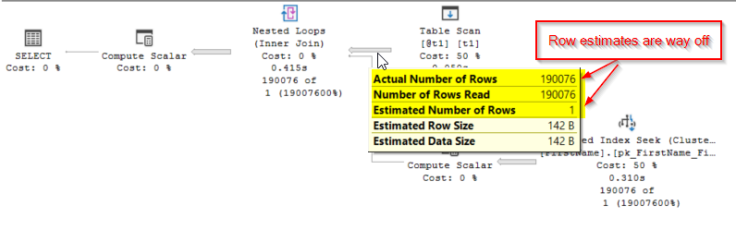
SELECT FirstNameId, FirstName

FROM ref.FirstName

SELECT \*

FROM @t1 t1

JOIN ref.FirstName ON FirstName.FirstNameId = t1.FirstNameId



Look at those estimates, SQL’s expecting only a single row from that table variable where as, in fact it’s going to read 190,076 rows.  There’s quite a difference there and one that’s probably causing SQL Server to pick an inefficient plan.

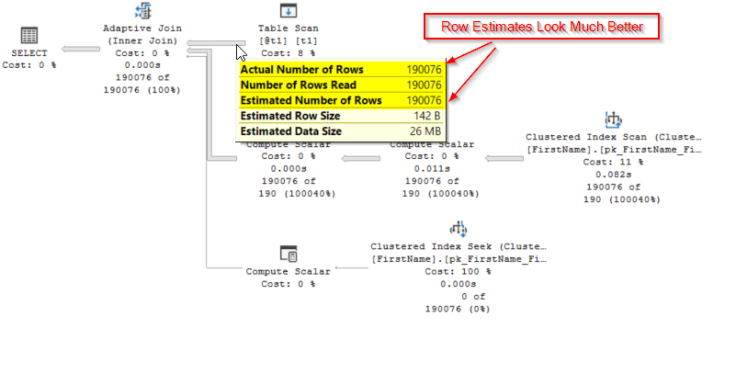
Lets just have a quick look at how many reads that query is doing…

***Table ‘FirstName’. Scan count 0, logical reads 570228***  
***Table ‘#A9FE9E6C’. Scan count 1, logical reads 554***

**So What’s The Deal With SQL 2019?**

The game changes massively with SQL 2019, or certainly with the current preview version, whether this makes it to the first release will remain to be seen.

In SQL 2019, SQL won’t just assume that a table variable is only going to include a single row.  In SQL 2019, SQL will actually wait until it knows how many rows are going to be in a table variable before it compiles the plan.  This means that SQL should have a much better idea of how many rows it’s expecting to see in a table variable and should be in a much better place to pick a good plan.  Let’s give that test query a go again but this time in SQL 2019 with a database with compatibility mode 150.



Things look a little different, aside from the fact that SQL has decided to now give us an adaptive join (a rather cool feature but not something that I want to cover in this post) which in this case is performing a hash match, those estimates now look much better.  This means that SQL Server should now be in a much better place to pick a more efficient plan.  So let’s have a look at what those page reads are looking like…

***Table ‘FirstName’. Scan count 0, logical reads 936***  
***Table ‘#B38808A’. Scan count 1, logical reads 554***

That’s a total of 1,490 reads as opposed to 570,782 reads with the old behaviour.  That’s a massive improvement.

***But What About OPTION (RECOMPILE)?***

There did used to be an old trick with table variables that you could get decent estimates by adding OPTION (RECOMPILE) to any query using a temp table, forcing it to recompile every time it ran but with correct row counts for table variables.

DECLARE @t1 TABLE (FirstNameId INT, FirstName VARCHAR(255))

INSERT INTO @t1

SELECT FirstNameId, FirstName

FROM ref.FirstName

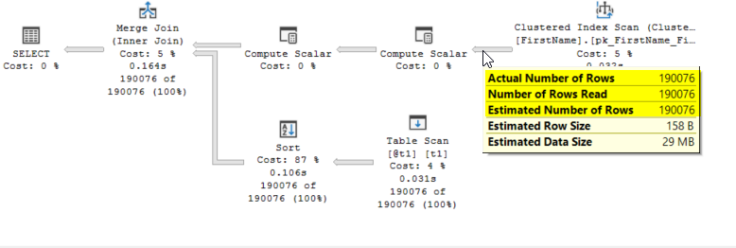
SELECT \*

FROM @t1 t1

JOIN ref.FirstName ON FirstName.FirstNameId = t1.FirstNameId

OPTION (RECOMPILE)

Lets just see how that compares back on our compat level 140 database…



Row estimates are bang on and interestingly something that I hadn’t expected, SQL Server has decided to sort the table variable, allowing it to perform a merge join.

But what about those page reads?

***Table ‘FirstName’. Scan count 0, logical reads 936***  
***Table ‘#BEF9BB52’. Scan count 1, logical reads 554***

Identical to our compat 150 query.

**So Which Is Better, Deferred Table Variable Compilation or OPTION (RECOMPILE)?**

The results may look the same but you need to remember that using OPTION(RECOMPILE) forces a plan to be recompiled with every execution and with all of the overhead that goes with it.  Deferred Compilation doesn’t have that problem, once the code is compiled, the plan is cached away and can be reused like any other cached plan, there’s no need to recompile.

While I think this is a great thing in 90% of cases, if you happen to have a query that gets a widely varying number of rows in the temp table, you may find yourself with a bad plan.

Another area where deferred compilation wins is that there’s no need to change code.  If you’ve got a bunch of badly performing procs riddled with table variables, your only real choice was to change the code.  You’d either have to add OPTION (RECOMPILE) or replace the table variable with a temp table.  Not always an attractive option.  Now with deferred compilation, as soon as the database compatibility level is changed to 150 it’ll just work with no code changes needed.

Now I’m certainly not saying that we all need to start jumping on the table variable band wagon but this does make them look a lot more of an attractive option than they have before although I’d want to do a lot more, careful testing of them before rolling them out in any sort of volume.

But, if you do tend to suffer with table variable induced performance issues and can’t realistically do anything about the code, this could be a big winner for you.