[**Query Plans: What Happens when Row Estimates Get High**](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/)

by [Kendra Little](http://www.brentozar.com/archive/author/kendra-little/) August 28, 2013

[7 comments](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comments)

FROM: <http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/>

SQL Server needs to have a reasonably correct estimate of how many rows your query is going to handle. If it doesn’t, things can go terribly, terribly wrong.

Most often, the problem is that SQL Server under estimates its work. In this case, the query optimizer thinks it has just a little bit of work to do, but in actuality it has much more. This results in “undersized” execution plans for the task– frequently a tiny little nested loop that’s run through over and over again while a larger join would do much better for the workload. This problem is often found in relation to the [Ascending Key Problem](http://blog.kejser.org/2011/07/01/the-ascending-key-problem-in-fact-tables-part-one-pain/) and out of date statistics in SQL Server.

But sometimes we have the opposite problem: SQL Server thinks it has a LOT of work to do, but really it only a little bit.

This pattern creates a totally different set of issues and has a very different impact on the workload of your SQL Server.

**A simple trick to inflate row estimates**

I’ve run into problems with workloads where overestimates are happening several times over the years, each time with completely different root causes. Recently I’ve found a simple way to reproduce the phenomena in AdventureWorks that makes it easy to see some of the impacts of overestimating a query’s work.

Let’s start by looking at the execution plan for the query behind [AdventureWorks2012](http://msftdbprodsamples.codeplex.com/releases/view/55330)‘s [HumanResources].[vEmployee] view. We’ll run this and get an actual execution plan as well as IO statistics.

[view source](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#viewSource)



[print](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#printSource)[?](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#about)

|  |  |
| --- | --- |
| SET STATISTICS IO,TIME ON | |
| GO |

|  |  |
| --- | --- |
| SELECT e.[BusinessEntityID], | |
| p.[Title], |

|  |
| --- |
| p.[FirstName], |
| p.[MiddleName], | |

|  |  |
| --- | --- |
| p.[LastName], | |
| p.[Suffix], |

|  |
| --- |
| e.[JobTitle], |
| pp.[PhoneNumber], | |

|  |  |
| --- | --- |
| pnt.[Name] AS [PhoneNumberType], | |
| ea.[EmailAddress], |

|  |  |
| --- | --- |
| p.[EmailPromotion], | |
| a.[AddressLine1], |

|  |  |
| --- | --- |
| a.[AddressLine2], | |
| a.[City], |

|  |  |
| --- | --- |
| sp.[Name] AS [StateProvinceName], | |
| a.[PostalCode], |

|  |  |
| --- | --- |
| cr.[Name] AS [CountryRegionName], | |
| p.[AdditionalContactInfo] |

|  |
| --- |
| FROM   [HumanResources].[Employee] AS e |
| INNER JOIN [Person].[Person] AS p | |

|  |
| --- |
| ON p.[BusinessEntityID] = e.[BusinessEntityID] |
| INNER JOIN [Person].[BusinessEntityAddress] AS bea | |

|  |  |
| --- | --- |
| ON bea.[BusinessEntityID] = e.[BusinessEntityID] | |
| INNER JOIN [Person].[Address] AS a |

|  |
| --- |
| ON a.[AddressID] = bea.[AddressID] |
| INNER JOIN [Person].[StateProvince] AS sp | |

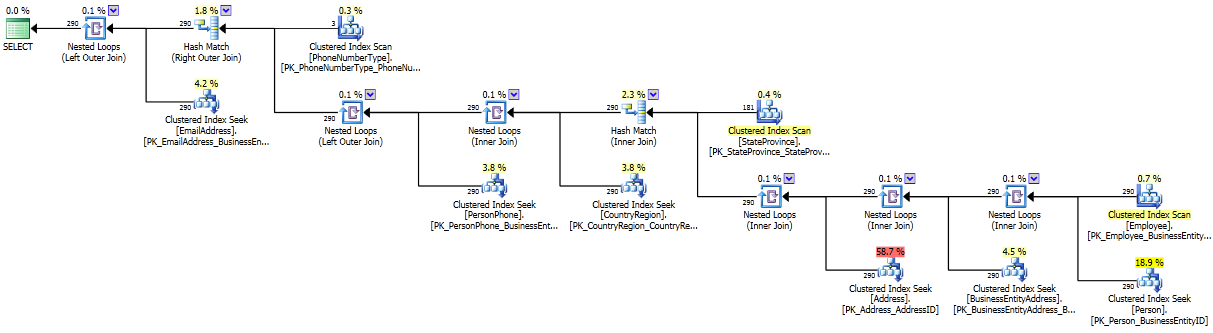
|  |  |
| --- | --- |
| ON sp.[StateProvinceID] = a.[StateProvinceID] | |
| INNER JOIN [Person].[CountryRegion] AS cr |

|  |  |
| --- | --- |
| ON cr.[CountryRegionCode] = sp.[CountryRegionCode] | |
| LEFT OUTER JOIN [Person].[PersonPhone] AS pp |

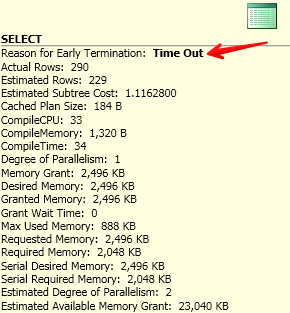
|  |
| --- |
| ON pp.BusinessEntityID = p.[BusinessEntityID] |
| LEFT OUTER JOIN [Person].[PhoneNumberType] AS pnt | |

|  |  |
| --- | --- |
| ON pp.[PhoneNumberTypeID] = pnt.[PhoneNumberTypeID] | |
| LEFT OUTER JOIN [Person].[EmailAddress] AS ea |

|  |  |
| --- | --- |
| ON p.[BusinessEntityID] = ea.[BusinessEntityID] | |
| GO |

Here’s the general shape of our plan. Just note that the bars connecting operators are narrow– SQL Server has pretty low (and accurate) estimates of how much data will be flowing through these pipes:  
[](http://www.brentozar.com/wp-content/uploads/2020/06/Plan-1.png)

Now, this query is already a little troubled. It has a lot of joins, including outer joins. It shows an optimization timeout in the plan. The plan estimated cost is 1.1 and it estimates it’ll bring back 229 rows. In my test environment it takes 16 ms to run and does several thousand logical reads across the many tables it joins. This means the query optimizer didn’t consider every possible plan, but hey, our runtime was pretty darn good anyway.

[](http://www.brentozar.com/wp-content/uploads/2020/06/Plan-1-Timeout1.png)

Now let’s cause some *real* trouble. We’re going to make only one type change to this query– we’re going to add functions to the joins. These functions mimic what a user might do if they were concerned about leading/trailing characters. This is going to do several things:

* Force SQL Server to do an implicit conversion to the columns in order to apply the functions
* Effectively apply the functions row-by-row to do the comparison at runtime
* Together this makes it incredibly hard for it to properly estimate the number of rows that come out of the joins.

Here’s our revised query:

[view source](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#viewSource)



[print](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#printSource)[?](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#about)

|  |  |
| --- | --- |
| SET STATISTICS IO, TIME ON; | |
| GO |

|  |  |
| --- | --- |
| SELECT e.[BusinessEntityID], | |
| p.[Title], |

|  |
| --- |
| p.[FirstName], |
| p.[MiddleName], | |

|  |  |
| --- | --- |
| p.[LastName], | |
| p.[Suffix], |

|  |
| --- |
| e.[JobTitle], |
| pp.[PhoneNumber], | |

|  |  |
| --- | --- |
| pnt.[Name] AS [PhoneNumberType], | |
| ea.[EmailAddress], |

|  |  |
| --- | --- |
| p.[EmailPromotion], | |
| a.[AddressLine1], |

|  |  |
| --- | --- |
| a.[AddressLine2], | |
| a.[City], |

|  |  |
| --- | --- |
| sp.[Name] AS [StateProvinceName], | |
| a.[PostalCode], |

|  |  |
| --- | --- |
| cr.[Name] AS [CountryRegionName], | |
| p.[AdditionalContactInfo] |

|  |
| --- |
| FROM   [HumanResources].[Employee] AS e |
| INNER JOIN [Person].[Person] AS p | |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(p.[BusinessEntityID])) = RTRIM(LTRIM(e.[BusinessEntityID])) | |
| INNER JOIN [Person].[BusinessEntityAddress] AS bea |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(bea.[BusinessEntityID])) = RTRIM(LTRIM(e.[BusinessEntityID])) | |
| INNER JOIN [Person].[Address] AS a |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(a.[AddressID])) = RTRIM(LTRIM(bea.[AddressID])) | |
| INNER JOIN [Person].[StateProvince] AS sp |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(sp.[StateProvinceID])) = RTRIM(LTRIM(a.[StateProvinceID])) | |
| INNER JOIN [Person].[CountryRegion] AS cr |

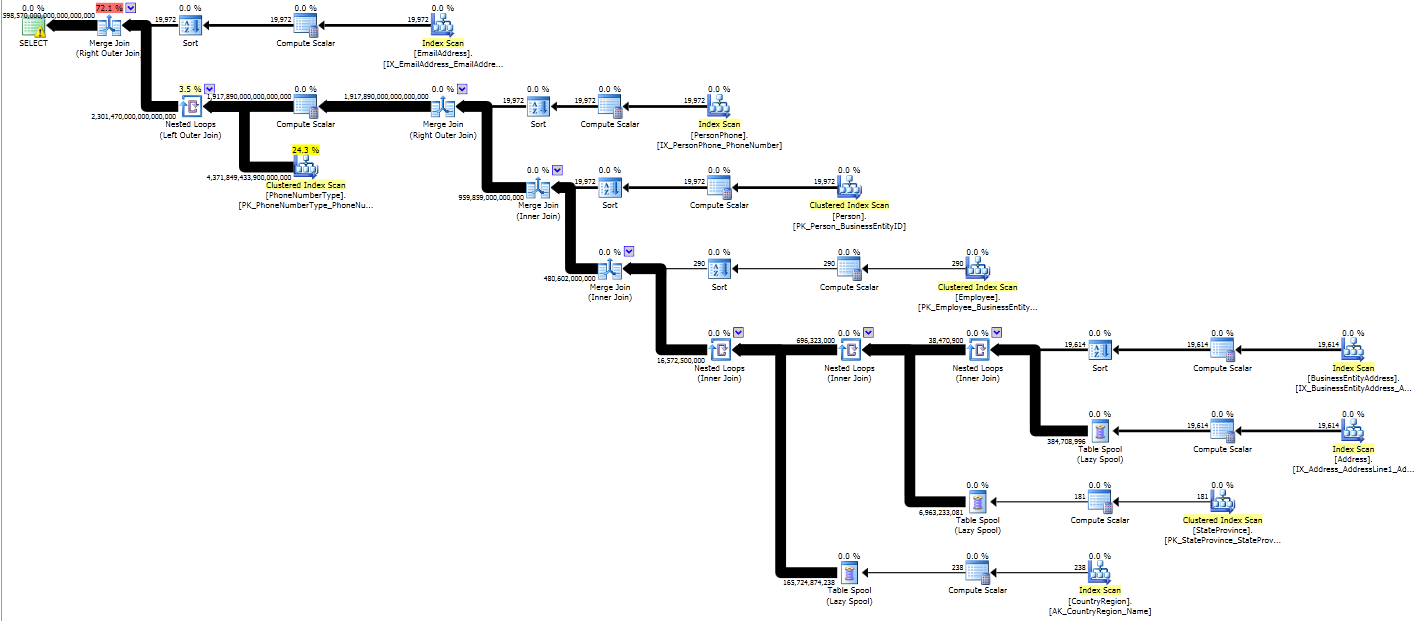
|  |  |
| --- | --- |
| ON RTRIM(LTRIM(cr.[CountryRegionCode])) = RTRIM(LTRIM(sp.[CountryRegionCode])) | |
| LEFT OUTER JOIN [Person].[PersonPhone] AS pp |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(pp.BusinessEntityID)) = RTRIM(LTRIM(p.[BusinessEntityID])) | |
| LEFT OUTER JOIN [Person].[PhoneNumberType] AS pnt |

|  |  |
| --- | --- |
| ON RTRIM(LTRIM(pp.[PhoneNumberTypeID])) = RTRIM(LTRIM(pnt.[PhoneNumberTypeID])) | |
| LEFT OUTER JOIN [Person].[EmailAddress] AS ea |

|  |
| --- |
| ON RTRIM(LTRIM(p.[BusinessEntityID])) = RTRIM(LTRIM(ea.[BusinessEntityID])) |

There have been a LOT of changes to the shape of this plan, as well as to the size of those pipes!

[](http://www.brentozar.com/wp-content/uploads/2020/06/Plan-2.png)

Instead of being able to use column or index statistics to properly estimate how many rows that come out of the joins, instead we end up with estimates that are sized as if every row might join with every row (a Cartesian product). That requires some heavy lifting!

Our second plan has an estimated cost of 645 trillion. SQL Server thinks it might return 4,598,570,000,000,000,000,000 rows (however many that is– I’m not sure how you even say that). The return data set is estimated as a HUGE amount of megabytes.

In my test environment it takes 122,046 ms to run (as compared to 16 ms) and does more than 4.8 million logical reads (as compared to several thousand). It chews up lots of CPU on three of my four virtual CPUs for the whole time it runs, even though it’s not parallel. Of course, it returns the same 290 rows of data at the end as the original query. (It just takes a LOT more work to do it!)

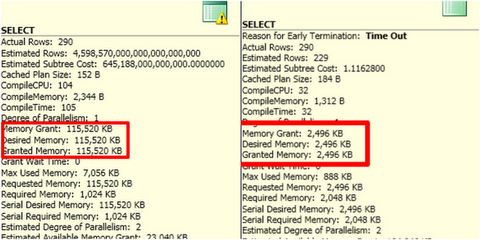
We can learn a lot from examining why this query has to use 7,600 times more CPU. And we can use this query to reproduce some pernicious problems that sometimes attack production environments.

**Big queries need big memory reservations**

Let’s compare some information from our two queries. We’re using SQL Server 2012 so we get some very rich information in our execution plans regarding our “Query Workspace Memory Grant”. This type of memory is used for query execution specifically. When SQL Server starts running a query it needs to figure out a minimum amount of memory that’s appropriate for all the sorts, spools, joins, and other things it may need to do in memory in the query. The more work it needs to do, the more of a query workspace memory it may need to ask for.

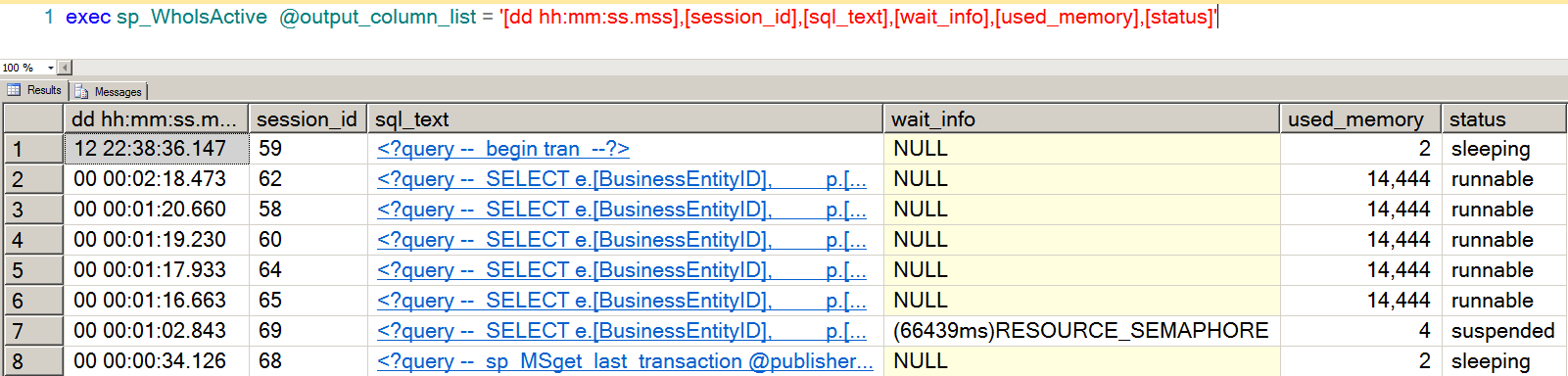
This memory is totally separate from the memory SQL Server uses to cache data in memory (the “buffer pool”) and the memory SQL Server uses to cache execution plans. In fact, this memory is “stolen” from the memory used for caching data pages. (Hey, it’s gotta come from somewhere, right?)

Our revised query (overly high estimates due to the functions in joins) is on the left. Our original query is on the right.

[](http://www.brentozar.com/wp-content/uploads/2020/06/Memory-Grant-Comparison.jpg)

That’s a big difference, right?

If you run a lot of big queries like this at the same time, those bigger query workspace memory grants mean that you may run low on the available amount of workspace memory for those types of queries. When this happens, new queries that come in may have to wait for a query workspace memory grant to run. This shows up as a RESOURCE\_SEMAPHOREwait, which is shown here with [Adam Machanic](http://sqlblog.com/blogs/adam_machanic/)‘s nifty[sp\_whoisactive stored procedure](http://brentozar.com/go/active).

[](http://www.brentozar.com/wp-content/uploads/2020/06/sp_whoisactive-Resource-Semaphore.png)

When this waits start happening, queries experiencing it literally can’t get out of the gate. They need a size of memory grant that just isn’t available, and they’ve got to wait around until their minimum grant can be met. (That’s fair, right? What wants to start running if there’s not enough memory to manage the query operators they need?)

*Aside:* Curious if you’re experiencing this problem in production? Our [sp\_blitz™](http://www.brentozar.com/blitz/) procedure helps detect if you’re experiencing this type of wait.

**Big queries may tear up tempdb**

Another thing we can see with this experiment is that some large-sized operators may be super-expensive, even on small row sets.

Exactly what made our runtime so bad? Sure we *thought* a ton of rows were possibly going to come out of those joins, but in actuality they didn’t. It turns out that the second query actually really DID have to do a lot more IO than the first query. Here’s our STATISTICS IO output:

[view source](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#viewSource)



[print](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#printSource)[?](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#about)

|  |
| --- |
| (290 row(s) affected) |
| Table 'Worktable'. Scan count 583, logical reads 4864543, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. | |

|  |
| --- |
| Table 'PhoneNumberType'. Scan count 1, logical reads 581, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |
| Table 'CountryRegion'. Scan count 1, logical reads 4, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

|  |
| --- |
| Table 'StateProvince'. Scan count 1, logical reads 4, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |
| Table 'Address'. Scan count 1, logical reads 216, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

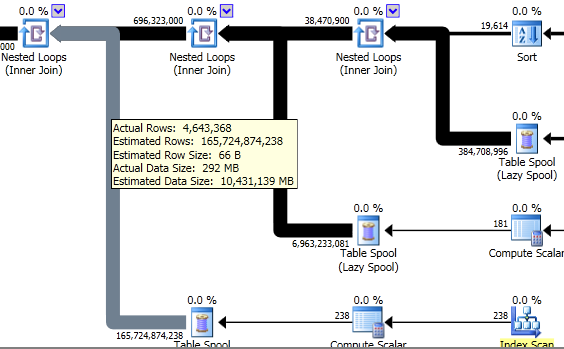
|  |
| --- |
| Table 'BusinessEntityAddress'. Scan count 1, logical reads 46, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |
| Table 'Employee'. Scan count 1, logical reads 9, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

|  |
| --- |
| Table 'Person'. Scan count 1, logical reads 3820, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |
| Table 'PersonPhone'. Scan count 1, logical reads 122, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

|  |
| --- |
| Table 'EmailAddress'. Scan count 1, logical reads 186, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

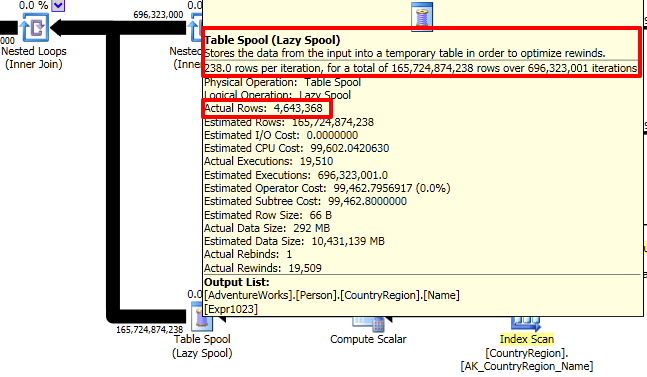
Look at that first line. Why the heck did we have to do almost 5 million reads against ‘Worktable’? And what exactly *is* worktable?

Let’s go back into that plan again. Look at the spool operators– they are where all those big pipes start. First we’ll hover over the line coming out of the spool.

[](http://www.brentozar.com/wp-content/uploads/2020/06/Plan-2-Spool-View-1.png)

These “spools” are worktables being built in memory (and tempdb) in SQL Server. Not only are there a high estimated number of rows coming out of these, but there’s a high *actual* amount of rows coming out of them as well.

If we look at the spool operator itself, we can see that each spool had to be accessed over and over again at runtime in our system to feed data back up into the nested loop join:

[](http://www.brentozar.com/wp-content/uploads/2020/06/Plan-2-Spool-View-2.png)

Because SQL Server had to do a lot of heavy row-by-row comparisons and wasn’t sure exactly how many might feed out of each of them, it decided to build “spool” temporary tables behind the scenes. It loaded the temp tables up in memory turn by turn and then had to go through a long, painful process of reading from them. It did this three times, in sequence.

**You don’t have to have big data to have big query problems**

The SQL Server optimizer is really clever, but often in production it gets into a bad situation. This can be caused by complex query patterns, poor data statistics, or TSQL anti-patterns.

This can be really tricky to identify, diagnose and fix in production if you’ve never seen it before. If you’re used to working with small databases, don’t worry– you can use techniques like I’ve outlined here to help you easily recreate in a small test environment what production might look like with big “beefy” queries running all at once.

[↑ Back to top](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#top)

7 comments

[Leave a comment](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comments)

1. http://1.gravatar.com/avatar/5f2e654f46aed2aacf9fbe1ccc975527?s=50&d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*William* [August 28, 2013 | 2:09 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-465901)

Very good info, Kendra! I’m walking through it a 2nd time.

FWIW, according to WolframAlpha.com, that humongous number is about 4 sextillion \*snicker\*.

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=465901#respond)

* + http://0.gravatar.com/avatar/a478f8bd3eaddbbcb15087896c2a58d2?s=50&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*Kendra Little* [August 28, 2013 | 7:27 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-466027)

Thank heavens, the internet can help me count! :)

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=466027#respond)

1. http://0.gravatar.com/avatar/0faaed902937cc2dd2df274553304d62?s=50&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*Brendan Morgan* [August 28, 2013 | 3:59 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-465958)

I normally run into the issue where SQL Server underestimates the number of rows when joining tables with a foreign key defined as referenced in <https://connect.microsoft.com/SQLServer/feedback/details/772232/make-optimizer-estimations-more-accurate-by-using-metadata>  
and turns even worse when using multiple columns in the join similiar to <https://connect.microsoft.com/SQLServer/feedback/details/243805/query-optimizer-choosing-improper-join-resulting-in-longer-execution-time>

I hadn’t heard of SQL Server overestimating rows returning so it was interesting to learn about this situation. Thanks Kendra.

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=465958#respond)

* + http://0.gravatar.com/avatar/a478f8bd3eaddbbcb15087896c2a58d2?s=50&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*Kendra Little* [August 28, 2013 | 7:29 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-466028)

It’s weird, isn’t it? I’ve run into it a couple of times where it’s all “ZOMG, that’s going to be a ton of work!” and it actually creates bizarre problems.

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=466028#respond)

1. http://0.gravatar.com/avatar/0e747083685b78a6fb3f264b48ab253d?s=50&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*tobi* [August 28, 2013 | 5:12 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-465988)

Those row counts might exhaust SQL Servers internal numeric precision ;-)Maybe we can induce negative rowcounts or “NaN rows”.

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=465988#respond)

*Todd Everett* [August 28, 2013 | 8:03 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-466048)

Great post! I had this exact problem once – those HUGE cumulative row estimates and bunches of merge scan joins when I knew there would be only a few rows actually returned. I wasn’t armed with any of the understanding as to why that you have provided here so this is great. I ended up breaking the query into 2 and using the first query find the needle rows in the hay stack and put them in a temp table. Then I joined that temp table to the world in the second query. Since the optimizer assumed 1 row for the temp table, it went back to low row estimates and indexed loop joins. That trick worked, but perhaps now better understanding the causes of inflated estimates I could fix the issue without breaking up the query.

[Reply](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/?replytocom=466048#respond)

1. http://0.gravatar.com/avatar/25da4274aac8109f1d0a011d23ccd5ca?s=50&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D50&r=G

*Randy* [August 28, 2013 | 9:20 pm](http://www.brentozar.com/archive/2013/08/query-plans-what-happens-when-row-estimates-get-high/#comment-466101)

I think we’re running into this. Tables with hundreds of millions of rows, dynamic ORM-generated sql with many joins, application users allowed to run LIKE queries, sometimes even double wildcard LIKE queries, etc. While we are getting better at identifying execution plans that are “bad” or inappropriate and using dbcc freeProcCache (planhandle) to remove them in response to performance complaints, I’m currently looking at our update stats jobs which run every night, sampling 80%. These use an algorithm to focus on stats needing the work most ( rows in table and changed rows ).

Recent reading is showing that unless you go below 50% sample rate (guaranteed disaster in our data base ), its faster to use fullscan ( and less thrashing of tempdb ). Update stat runs multi-threaded in fullscan. So now I just have to convince management to let me make some changes.