**A Self-Tuning Fill Factor Technique for SQL Server – Part 1**

[Mike Byrd](https://www.sqlservercentral.com/author/Mike-Byrd), 2019-08-20

FROM: <https://www.sqlservercentral.com/articles/a-self-tuning-fill-factor-technique-for-sql-server-part-1?preview_id=3668579>

Introduction

So what’s all the fuss about Fill Factor? It is a SQL Server parameter I’ve ignored for 20+ years. The main reason was/is I’ve had no idea about what value to use. There is just no documented definitive guidance.

Back in April at [SQL Saturday #830 - Colorado Springs](http://www.sqlsaturday.com/830/eventhome.aspx), I attended two sessions by Jeff Moden, titled [“Black Arts” Index Maintenance: How the “Best Practice” Methods are Silently Killing Performance](https://www.sqlsaturday.com/830/Sessions/Details.aspx?sid=88238). These sessions were outstanding and are downloadable from the SQL Saturday web site (<https://www.sqlsaturday.com/830/Sessions/Schedule.aspx>). A Reader’s Digest version of his presentations is as follows: contrary to the popular concept to reorganize at 10% fragmentation and rebuild indexes at 30%. He showed (in the 2 presentations) that page splits caused tremendous performance degradation, and you should probably rebuild indexes at 1% fragmentation. I don’t want to steal his “thunder” (and it is forthcoming in another series by him).

But on the flight back home, I got to thinking about everything he said and how fill factor also “factors” into index performance and maintenance. It got me to thinking about a brute force optimization technique I implemented way back when I worked at the Air Force Rocket Propulsion Laboratory. Yes, I was once a rocket scientist and still a geek! It was a technique that involved tweaking various parameters (plus and minus percentages) and reducing the tweaks to an “optimum” solution. This optimization technique was not always perfect as in a multi-dimensional scenario there are valleys that this solution could drill down on, but would offer “near” optimum solutions that let us explore options.

Frankly speaking, what happened is somewhat amazing. Using the “still experimental” methods that I explain in this article on a client’s system, the end result is that the overall database wait times showed improvement (decreases) of about 30%. That’s not a trivial improvement.  I hope this article will serve as an impetus for others to try these and other experiments on their indexes to come up with other improvements they might wish to share with the community at large.

Part 1 of this series will include an Overview and Code Details (including the actual TSQL script). Part 2 will cover an analysis of the code and its results over a 3 month period as well as performance details and sections on performance, current limitations, and anticipated future improvements.

Overview

What was needed was to collect index parametrics and look for patterns. This involved using the sys.dm\_db\_index\_physical\_stats and sys.dm\_db\_index\_operational\_stats views and capturing these index parametrics before and after each index rebuild. Prior to this task, my choice for index fragmentation was a modified version of Ola Hallengren’s defrag script that is configured to only start rebuilds at 10% and no reorganizes. For this project, this script was modified to capture and store (in a table) the before and after parametrics for each index rebuild  within a specified database.

Initially I started looking at the top 15 average fragmentation indexes greater than 1.0%, but eventually changed it to 1.2% because of logical fragmentation issues. In the meantime, if a fill factor has not established, the previous fill factor (for that index) is decremented and the resulting average fragmentation compared to the previous average fragmentation. All indexes prior to this task had 100% fill factors.

This process is repeated every 24 hours in a SQL Agent job until the index’s new average fragmentation is greater than the previous one. The previous fill factor is notated and fixed for that specific index. A secondary look at the data over 90 days (since the fill factor was established), inserts an index for a once again review (data collection). While this may temporarily degrade that index’s performance it ensures that a once again “near optimum” fill factor can be established taking into account any new data skew and application utilization of the database.

Details

This process (index rebuild and fill factor perturbation) was incorporated into a SQL Agent job that runs daily during a period of low server activity. It stores data from the SQL Agent job and inspects that data for possible fill factor adjustments. The following discussion describes the attached script in detail.  I recommend turning on Line Numbering in SSMS and having the script listed in a second window for following this discussion.

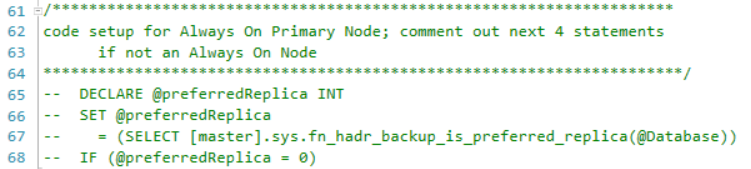
Line 4 is the specification of the database name. Currently the script is setup for only one database, however, coding within the working tables will allow the script to eventually be expanded to multiple databases within a server. If this line remains uncommented the script will use the current database.

https://www.sqlservercentral.com/wp-content/uploads/2019/08/img_5d46d84fdfa34.png

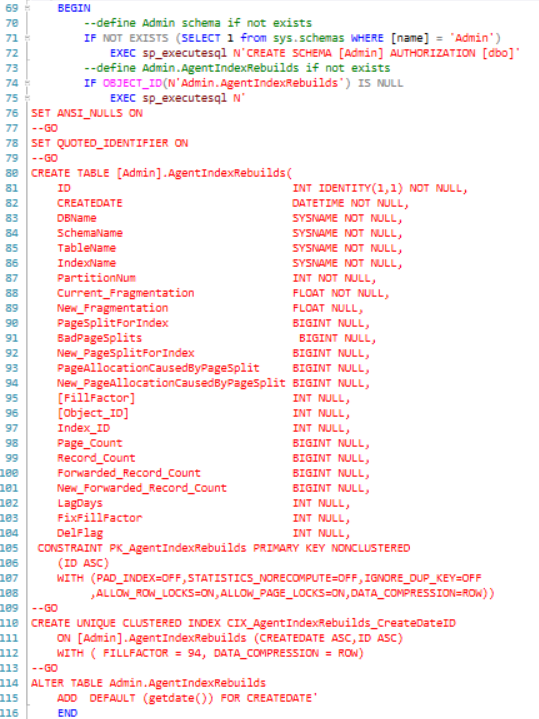
Line 60 gets the name of the current database.

https://www.sqlservercentral.com/wp-content/uploads/2019/08/img_5d46d8bb2e4aa.png

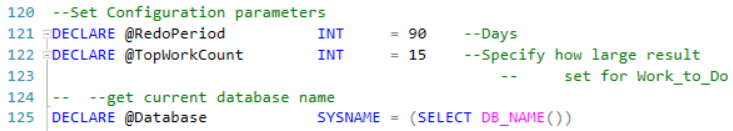
The commented code in lines 65-68 should be uncommented when used in an Always On environment.



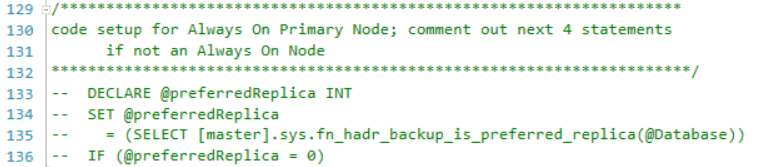
Lines 70-116 define the Admin schema and also the Admin.AgentIndexRebuilds table. This table has evolved during the study to this final version. Not all parameters are used to determine each index’s fill factor, but I thought (and still do) that too much data collection allows sufficient analysis rather than not enough. Note that the database name is retained so that this code can be extended to multiple DBs on the same server. Variables prefaced with “New” are the results after the particular index rebuild has completed (with or without a new fill factor). There will be more discussion on this table later in the Analysis section (Part 2).



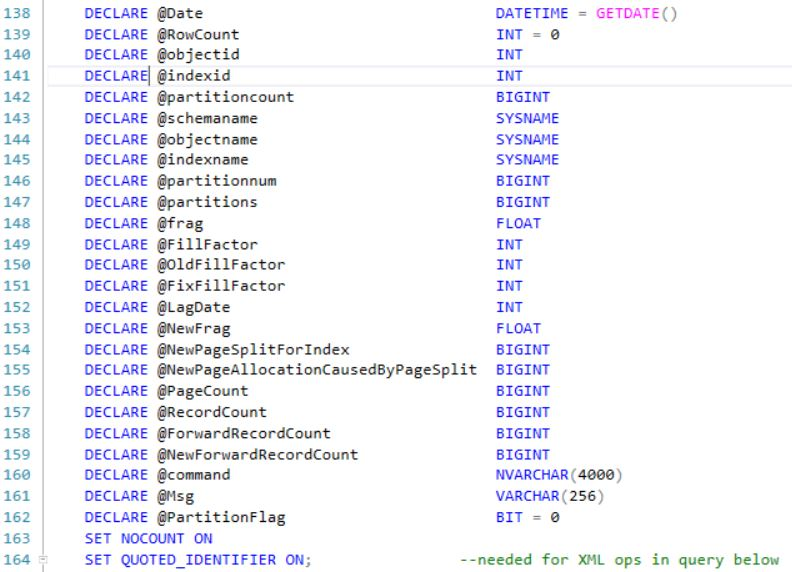
Lines 120-125 are the primary configuration parameters for this script. **@RedoPeriod** is the number of days to wait until an identified index is re-evaluated (more on this in the code discussion below). **@TopWorkCount** is a specified number for the **TOP** command to return the specified number of most fragmented rows. Initially I started with a value of 15, but within 3 weeks most of the time the result set returned less than 15 rows (this was primarily due to the improved fill factor and decreased split pages). **@Database** is the current database name.



The commented code in lines 133-136 should be uncommented when used in an Always On environment.



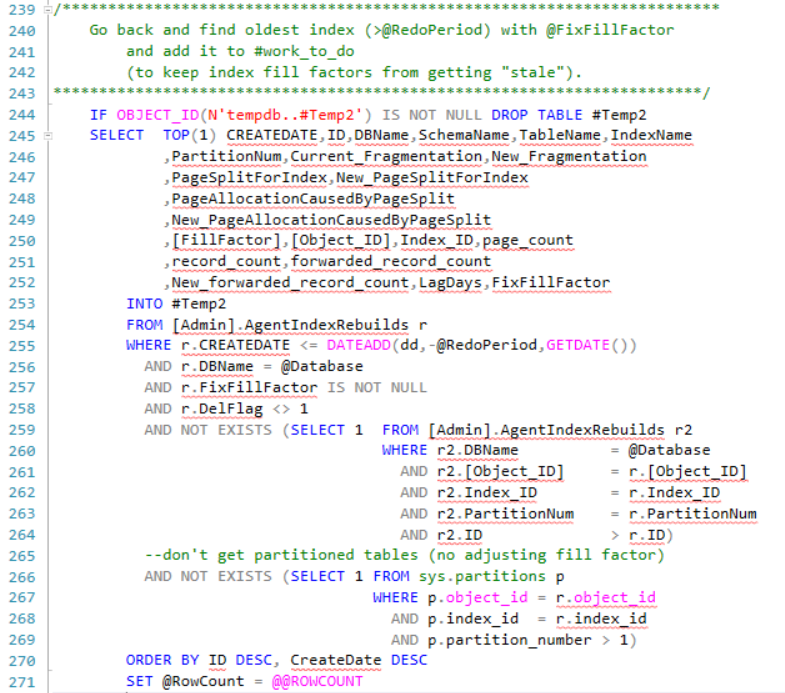
Lines 138-162 are variable declarations used in the subsequent code.



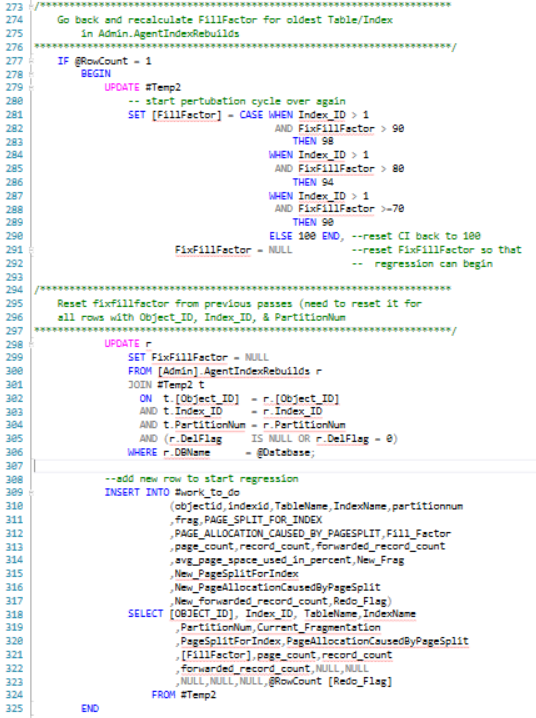
Lines 171-237 is the code used to select the TOP worse fragmented and split page indexes at the current time. The query calls sys.dm\_db\_index\_physical\_stats to get the current fragmentation and sys.dm\_db\_index\_operational\_stats to get page splits as well as sys.indexes and sys.objects to get table and index data. This data is put into a working table (#work\_to\_do) for subsequent utilization. This query is dual-functional; it always does an index rebuild (for the specified index) as a de facto defrag, but also a fill factor perturbation if not already fixed. If you examine the ORDER BY clause it is also dual-functional – even days the TOP n rows are returned by maximum average fragmentation descending and on odd days the TOP n rows are returned by maximum page-splits per index descending.



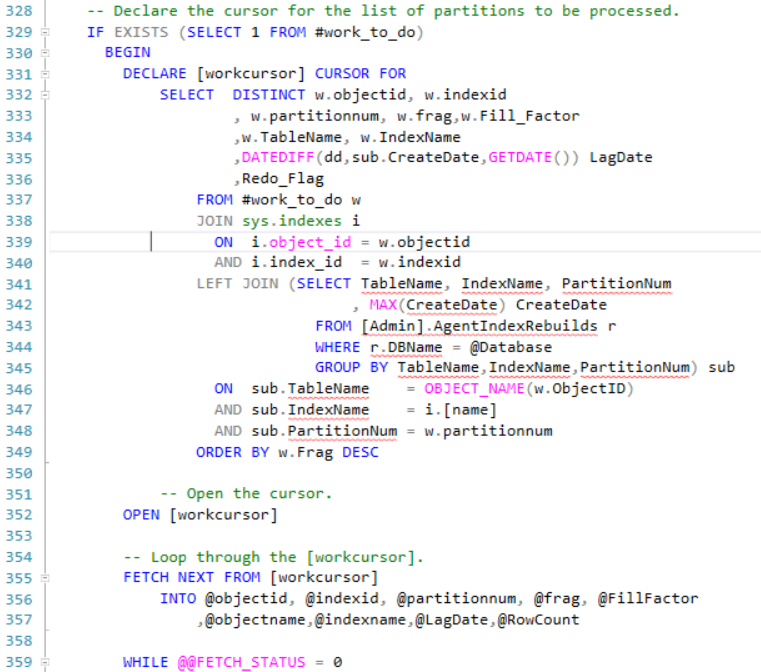
Lines 239-271 review the existing data to see if there is a candidate index for re-evaluation of its fill factor (where the fill factor is already set).



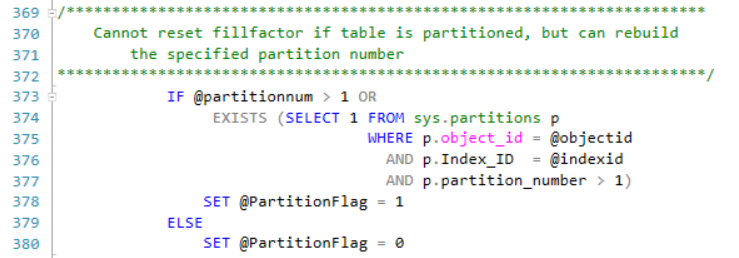
If there is, a row is added to #work\_to\_do for that index and the **@Redo\_Flag** is set to one. As configured, the **@RedoPeriod** is 90 days – this looks at any existing index in the Admin.AgentIndexRebuilds table to see if last adjustment > **@RedoPeriod** .



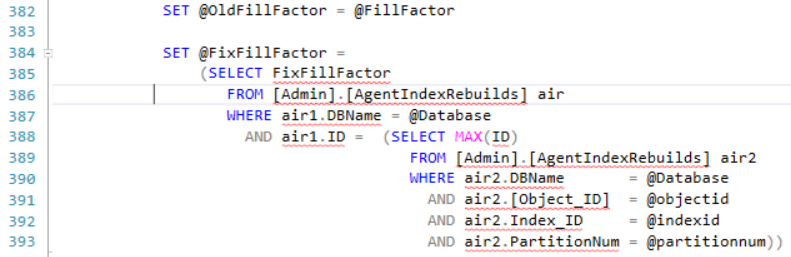
Lines 328-359 define the cursor used to rebuild each table/index in #work\_to\_do.



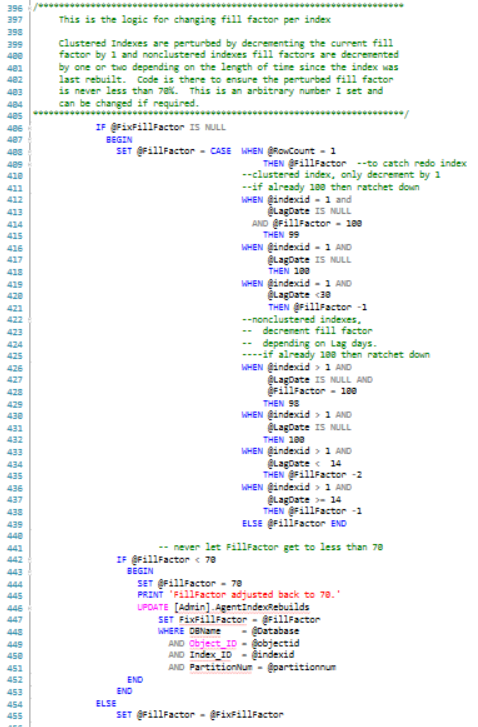
Lines 369-380 set up a flag to specify if an index is partitioned or not. Since you cannot change fill factor for a single partition within a partitioned index, the flag signals the code to only rebuild that particular index partition without a perturbation of the fill factor.



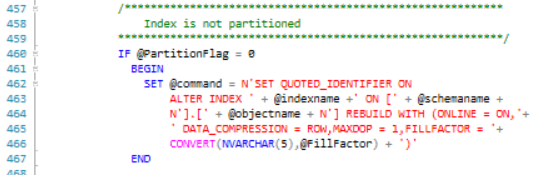
Lines 382-393 test to see if the fill factor is already fixed or not.



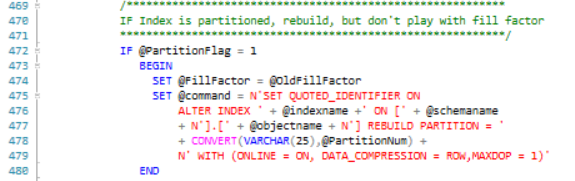
Lines 396-455 contain the logic for perturbing the fill factor. The code is very readable. Clustered Indexes are perturbed by decrementing the current fill factor by 1 and Nonclustered indexes fill factors are decremented by one or two, depending on the length of time since the index was last rebuilt. Code is there to ensure the perturbed fill factor is never less than 70%. This is an arbitrary number I set and can be changed if required.



Lines 436-443 setup the dynamic SQL to rebuild the index with the specified fill factor if the table is not partitioned.



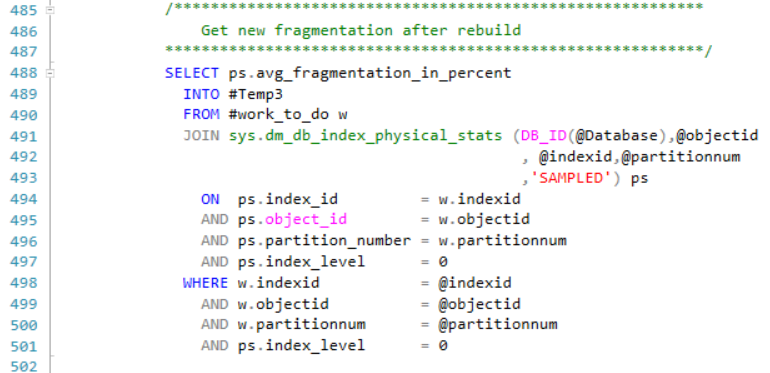
Lines 469-480 set up the dynamic SQL to rebuild the specified index for partitioned indexes. The Fill Factor is not changed for partitioned indexes.



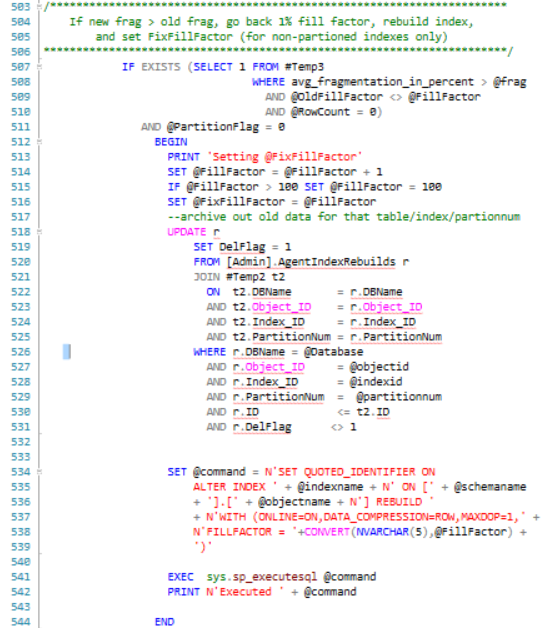
Lines 482-483 executes the generated dynamic SQL.

https://www.sqlservercentral.com/wp-content/uploads/2019/08/img_5d46f2a02d1a7.png

Lines 485-501 gets the new index fragmentation.



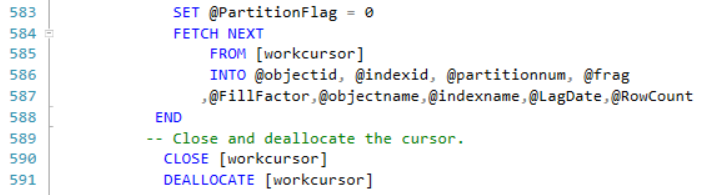
Lines 503-544 contain the logic to determine if the current fill factor causes more fragmentation. If so, the fill factor is backed up (decremented by 1), specified as fixed, and rebuild with the fixed fill factor.



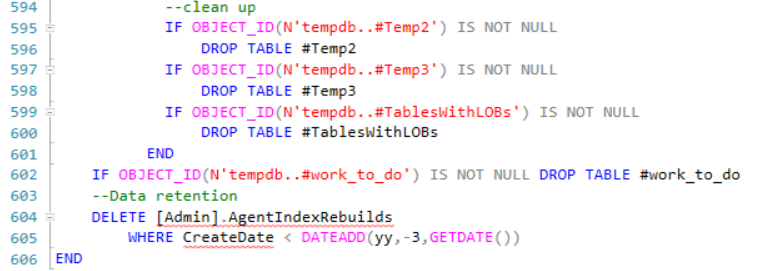
Lines 546-581 inserted the old and new index parametrics into the [Admin].AgentIndexRebuilds table for subsequent review by the next script run.



Lines 583-588 take care of some administrative book-keeping and fetch the data for the next cursor loop.



Lines 594-606 are just clean-up code.



Requirements

This code was developed and tested with SQL Server 2017 Enterprise Version. However, it is applicable to all versions of SQL Server from 2012 and upward.

It can also be used for Standard Editions (same versions) except the WITH Statement in the Index Rebuilds needs to have the “ONLINE = ON, Data\_Compression = ROW,” statement removed (3 occurrences). Data\_Compression = ROW option can (and should be) retained in SS 2016, SP 2 and upward for Standard Edition. In the event you are using Standard Edition consider scheduling this task when there is minimal database activity as without the ONLINE option you may have schema locking during the index rebuilds.

Otherwise, there are no other restrictions that I am aware of.

Summary

This whole project started as a proof of concept and as most POCs has made its way into production. While I agree that it’s not complete and isn’t perfect, it did result in a substantial decrease in wait time on a real system. It also demonstrates that effective automatic determination of Fill Factor **IS** possible.

I look forward to constructive critique and suggestions for improvements from the SQL Server Community.

Cheers,

Mike

Resources

* [SQLAgentScriptRebuildIndexes.txt](https://www.sqlservercentral.com/wp-content/uploads/2019/08/SQLAgentScriptRebuildIndexes.txt)

**A Self-Tuning Fill Factor Technique for SQL Server – Part 2**

[Mike Byrd](https://www.sqlservercentral.com/author/Mike-Byrd), 2019-08-27

Introduction

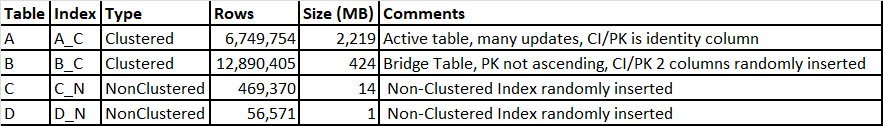
Fill Factor is a parameter most of us have ignored because there is no definitive way to calculate what the correct value should be. [Part 1](https://www.sqlservercentral.com/?p=3668579&post_type=ssc_article&preview_id=3668579) of this series describes a new technique to dynamically determine index fill factors for all indexes within a database. Part 2 covers an analysis of the data collection and its results over a 3 month period as well as performance details and sections on performance, current limitations, and anticipated future improvements.

Frankly speaking, what happened is somewhat amazing. Using the “still experimental” methods that I explain in this article on a client’s system, the end result is that the overall database wait times showed improvement (decreases) of about 30%. That’s not a trivial improvement, and I hope this article will serve as an impetus for others to try these and other experiments on their indexes to come up with other improvements they might wish to share with the community at large.

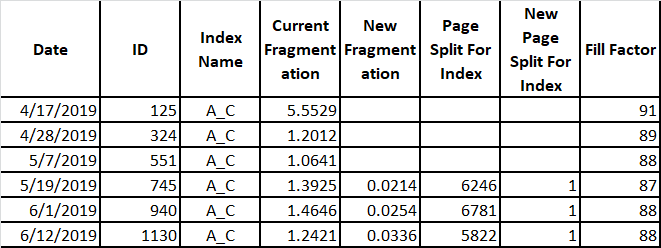
Analysis

Initially, this project started mainly as data collection to ascertain patterns in the index parametrics. As discussed in Part 1, sys.dm\_db\_index\_physical\_stats, sys.dm\_db\_index\_operational\_stats, sys.indexes, and sys.objects are used to gather parametrics for subsequent storage in the table [Admin].AgentIndexRebuilds. In this section, we will look at some of the patterns encountered.

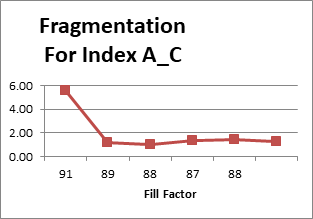
The testing was done on a very active OLTP production database which we will call TestDB (to protect my client’s confidentiality). TestDB contains ~225gb of data (real data, not stored procedures, views, etc.), 249 tables, and 694 indexes in a 24x7 environment. There are few hard data deletes as most of the data is soft-deleted (DelFlag). Only 164 indexes exceeded the average fragmentation specification (1.2%) and were recorded within the AgentIndexRebuilds table (over a 3 month period). While each index’s pattern was examined, in the interest of brevity, we will only consider the following typical indexes (names changed to protect client):



The A\_C index is a Clustered Index/Primary Key based on an identity column. Data is regularly added to it through the day (24 hours). Data collected for this index is shown below:

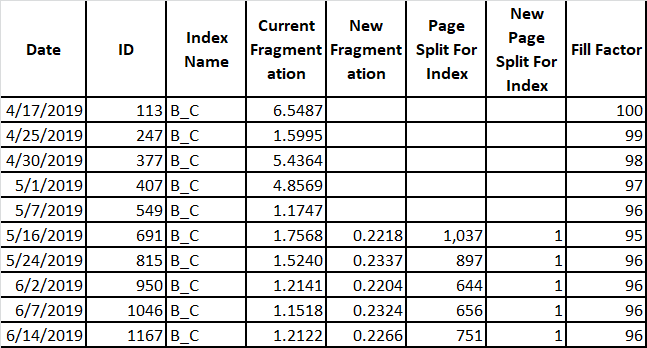


Missing data is from new columns added to table after data collection started. The value of 1 for New Page Split For Index was puzzling at first. Upon further investigation, it appears that after a rebuild it always starts with a value of 1. The bad news is that the “Page Split For Index” parameter yields both good and bad page splits. (A good explanation of good and bad page splits can be found at <http://www.sqlballs.com/2012/08/how-to-find-bad-page-splits.html>.) A little research indicates that extended events may be able to break out the bad page splits. Plotting out Current Fragmentation vs Fill Factor gives:

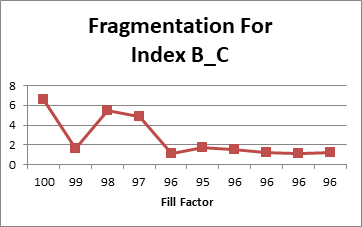


This trend in above figure is very typical. Fill Factor usually decreases until a minimum is met. In this case the code subsequently tried an 87 fill factor, but since the resulting fragmentation was larger than before the fill factor was fixed at 88%. The second 88 fill factor is from a subsequent index rebuild after the fill factor was fixed.

Looking at another clustered index B\_C, this one is in a bridge table where rows are continuously inserted (out of order). Looking at the data we get:

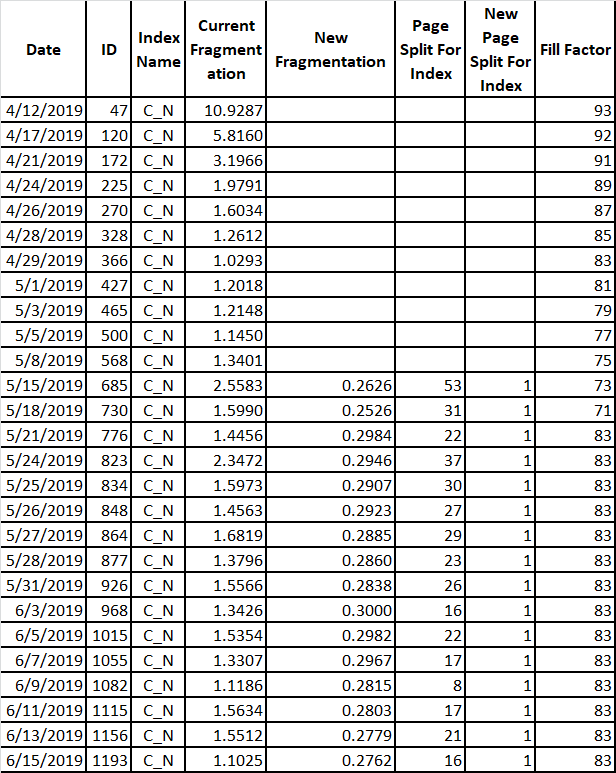


The missing data is from new columns added to table after data collection started. Plotting out Current Fragmentation vs date gives:

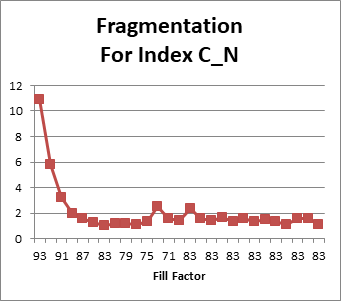


This is also a typical trend for the Clustered Index. The only data glitch was the dip in fragmentation at 99 fill factor. This would have normally been selected as the fixed fill factor, but the select code (as before) was not in place. Thus 96 was selected as the fixed fill factor. I’m not sure how to handle this condition going forward; any ideas from the readers would be appreciated.

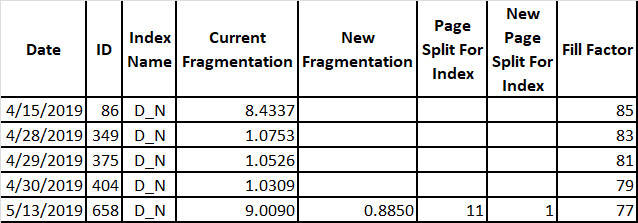
Looking at non-clustered index C\_N:



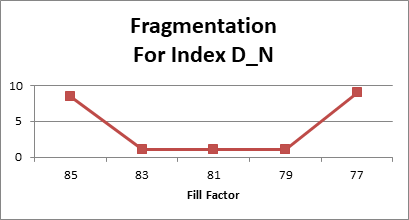
Again the code was not in place to fix the fill factor; hence the empty data slots. In this case, the fill factor was set at 83 after an extended excursion below that. What is really interesting is the data hopping around once the fill factor is set. Doesn’t seem so much to be a factor of new page splits, but more so of logical fragmentation (more than one index using same extent). There is no easy way to mitigate the logical fragmentation so I’ll leave that to someone else. As before, the fragmentation vs fill factor trend for this index is similar to before:



And finally, another non-clustered index example is D\_N and collected data for it is shown below:



In this specific case, had the fix fill factor code been in place to catch the 4/28 case, the fill factor should have been set to 83. But you can see that it continues to decrease some. Not sure which is best, 83, 81, or 79, but at least we are in the ball park and the fill factor is not 100 as it started out to be. Looking at the graph for this index we get

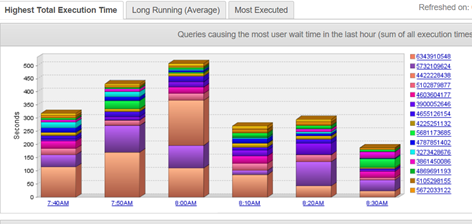


Obviously, the optimum fill factor for the current data is between 79-83%. In this particular case I would want to err on the bigger fill factor to reduce the number of logical IOs on the index. This is what the code currently does.

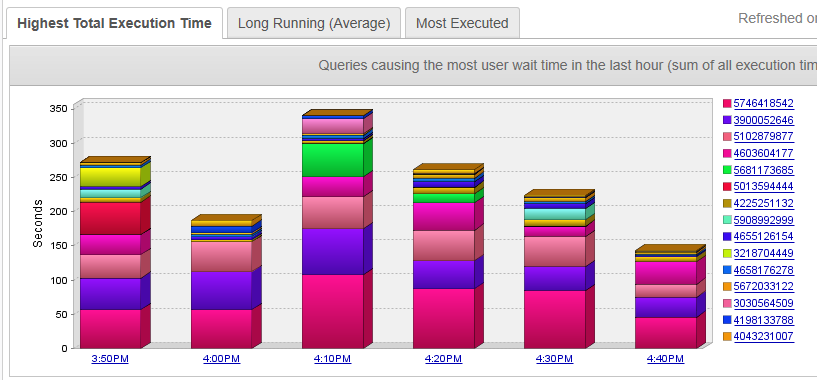
There are other instances (after studying Admin.AgentIndexRebuilds) that the code just doesn’t handle. I will eventually figure out what works, but now I review the table weekly to see what indexes the code is not catching.

For a short time, I considered using a least squares quadratic match to the data to determine best fill factor, but disregarded it to keep the current design as simple as it is.

Unfortunately I do not have hard numbers to measure performance improvements. But I did have some screen shots from the free version of DPA (Solarwinds Database Performance Analyzer).  The first was taken late Feb 2019



and the second from June 2019 (after 3 months of Fill Factor adjustment):



Notice that the vertical axis decreased from 500 seconds to 350 seconds (yes I am aware that these Screenshots were taken at different time of the day, but are representative of the entire day’s work). Each of the screen shots were very typical throughout the day for periods referenced. DPA showed an overall decrease in database wait times of about 30%. This is a significant number. I attribute it to 2 factors:

1. With decreased fragmentation, there are less page splits to wade through for IO range searches.
2. With improved fill factor, there are less bad page splits. The cost of a page split can be significant. First the new page has to be identified (most likely on a new extent), pointers made to it, data movement from the old page to the new page and finally all of this is logged.

Requirements

This code was developed and tested with SQL Server 2012 Enterprise Version. However, it is applicable to all versions of SQL Server from 2012 and upward. It can also be used for Standard Editions (same versions) except the WITH Statement in the Index Rebuilds needs to have the “ONLINE = ON, Data\_Compression = ROW,” statement removed (3 occurrences). Data\_Compression = ROW option can (and should be) retained in SS 2016, SP 2 and upward for Standard Edition.

In the event you are using Standard Edition consider scheduling this task when there is minimal database activity as without the ONLINE option you will have schema locking during the index rebuilds. Otherwise, there are no other restrictions that I am aware of.

Caveats

This methodology is not complete, but does represent a start on determining fill factor. Hopefully, it will evolve (along with the community’s help) to represent a much better product than it currently is. Some of the features (not in any particular priority and not to be considered complete) that still need to addressed are:

* Add code for multiple databases on same server; this may involve creating a new Maintenance database to retain the index parametrics table.
* Finalize and catch all edge conditions not currently accounted for
* Research extended events to identity “bad” page splits (already included in latest code revision).
* Rewrite code (current code is from proof of concept) – I intend to clean it up in the near future
* Current fill factor approach not implemented for partitioned indexes (it would really be nice to have fill factor per partition; that way the old data partitions could be set to 100% and the active partitions set as needed {this should be placed on Microsoft’s wish list} ).
* Try and build in additional features as suggested by the community
* More testing on performance improvements with firm numbers.
* Need better way of monitoring Admin.AgentIndexRebuilds – both manually and also self-reviewing. Am looking for a volunteer to implement a dash board.
* Need to get code into GitHub (still working on it; look for user MByrdTX)

Conclusion

This whole project started as a proof of concept and as most POCs made its way into production. While I agree that it’s not complete and isn’t perfect, it did result in a substantial decrease in wait time on a real system. It also demonstrates that effective automatic determination of Fill Factor **IS** possible.

My intention from the start was to make this available to the SQL Server Community without any restrictions. I look forward to constructive critique and suggestions, and hope that this methodology works its way into SQL Server mainstream.

Cheers,

Mike