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

By Mike Byrd

ByrdNest Consulting

# 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 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):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table** | **Index** | **Type** | **Rows** | **Size (MB)** | **Comments** |
| A | A\_C | Clustered | 6,749,754 | 2,219 | Active table, many updates, CI/PK is identity column |
| B | B\_C | Clustered | 12,890,405 | 424 | Bridge Table, PK not ascending, CI/PK 2 columns randomly inserted |
| C | C\_N | NonClustered | 469,370 | 14 | Non-Clustered Index randomly inserted |
| D | D\_N | NonClustered | 56,571 | 1 | Non-Clustered Index randomly inserted |

Table : Sample Indexes

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:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Date** | **ID** | **Index Name** | **Current Fragmentation** | **New Fragmentation** | **Page Split For Index** | **New Page Split For Index** | **Fill Factor** |
| 4/17/2019 | 125 | A\_C | 5.5529 |  |  |  | 91 |
| 4/28/2019 | 324 | A\_C | 1.2012 |  |  |  | 89 |
| 5/7/2019 | 551 | A\_C | 1.0641 |  |  |  | 88 |
| 5/19/2019 | 745 | A\_C | 1.3925 | 0.0214 | 6246 | 1 | 87 |
| 6/1/2019 | 940 | A\_C | 1.4646 | 0.0254 | 6781 | 1 | 88 |
| 6/12/2019 | 1130 | A\_C | 1.2421 | 0.0336 | 5822 | 1 | 88 |

Table : Index A\_C Data

Missing data is from new columns added to Table 2 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:

Figure : Fragmentation for Index A\_C

The trend in Figure 1 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:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Date** | **ID** | **Index Name** | **Current Fragmentation** | **New Fragmentation** | **Page Split For Index** | **New Page Split For Index** | **Fill Factor** |
| 4/17/2019 | 113 | B\_C | 6.548705 |  |  |  | 100 |
| 4/25/2019 | 247 | B\_C | 1.599545 |  |  |  | 99 |
| 4/30/2019 | 377 | B\_C | 5.436387 |  |  |  | 98 |
| 5/1/2019 | 407 | B\_C | 4.856919 |  |  |  | 97 |
| 5/7/2019 | 549 | B\_C | 1.174713 |  |  |  | 96 |
| 5/16/2019 | 691 | B\_C | 1.756837 | 0.221820894 | 1037 | 1 | 95 |
| 5/24/2019 | 815 | B\_C | 1.523979 | 0.233713691 | 897 | 1 | 96 |
| 6/2/2019 | 950 | B\_C | 1.214119 | 0.220436191 | 644 | 1 | 96 |
| 6/7/2019 | 1046 | B\_C | 1.151839 | 0.232396003 | 656 | 1 | 96 |
| 6/14/2019 | 1167 | B\_C | 1.212156 | 0.226648484 | 751 | 1 | 96 |

Table : Index B\_C Data

Missing data is from new columns added to Table 3 after data collection started. Plotting out Current Fragmentation vs date gives:

Figure : Index B\_C

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:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Date** | **ID** | **Index Name** | **Current Fragmentation** | **New Fragmentation** | **Page Split For Index** | **New Page Split For Index** | **Fill Factor** |
| 4/12/2019 | 47 | C\_N | 10.92865 |  |  |  | 93 |
| 4/17/2019 | 120 | C\_N | 5.816024 |  |  |  | 92 |
| 4/21/2019 | 172 | C\_N | 3.196622 |  |  |  | 91 |
| 4/24/2019 | 225 | C\_N | 1.979109 |  |  |  | 89 |
| 4/26/2019 | 270 | C\_N | 1.603421 |  |  |  | 87 |
| 4/28/2019 | 328 | C\_N | 1.261167 |  |  |  | 85 |
| 4/29/2019 | 366 | C\_N | 1.029336 |  |  |  | 83 |
| 5/1/2019 | 427 | C\_N | 1.201803 |  |  |  | 81 |
| 5/3/2019 | 465 | C\_N | 1.214772 |  |  |  | 79 |
| 5/5/2019 | 500 | C\_N | 1.145038 |  |  |  | 77 |
| 5/8/2019 | 568 | C\_N | 1.340111 |  |  |  | 75 |
| 5/15/2019 | 685 | C\_N | 2.558348 | 0.262582057 | 53 | 1 | 73 |
| 5/18/2019 | 730 | C\_N | 1.598963 | 0.252631579 | 31 | 1 | 71 |
| 5/21/2019 | 776 | C\_N | 1.445578 | 0.298359025 | 22 | 1 | 83 |
| 5/24/2019 | 823 | C\_N | 2.347188 | 0.29455081 | 37 | 1 | 83 |
| 5/25/2019 | 834 | C\_N | 1.597289 | 0.290697674 | 30 | 1 | 83 |
| 5/26/2019 | 848 | C\_N | 1.456311 | 0.292255236 | 27 | 1 | 83 |
| 5/27/2019 | 864 | C\_N | 1.681884 | 0.288461538 | 29 | 1 | 83 |
| 5/28/2019 | 877 | C\_N | 1.379638 | 0.285986654 | 23 | 1 | 83 |
| 5/31/2019 | 926 | C\_N | 1.556604 | 0.283822138 | 26 | 1 | 83 |
| 6/3/2019 | 968 | C\_N | 1.342616 | 0.3 | 16 | 1 | 83 |
| 6/5/2019 | 1015 | C\_N | 1.535414 | 0.298210736 | 22 | 1 | 83 |
| 6/7/2019 | 1055 | C\_N | 1.330705 | 0.296735905 | 17 | 1 | 83 |
| 6/9/2019 | 1082 | C\_N | 1.118568 | 0.281531532 | 8 | 1 | 83 |
| 6/11/2019 | 1115 | C\_N | 1.563372 | 0.280269058 | 17 | 1 | 83 |
| 6/13/2019 | 1156 | C\_N | 1.551247 | 0.277932185 | 21 | 1 | 83 |
| 6/15/2019 | 1193 | C\_N | 1.102536 | 0.276243094 | 16 | 1 | 83 |

Table : 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 the 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 (Figure 3).

Figure : Fragmentation for Index C\_N

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Date** | **ID** | **Index Name** | **Current Fragmentation** | **New Fragmentation** | **Page Split For Index** | **New Page Split For Index** | **Fill Factor** |
| 4/15/2019 | 86 | D\_N | 8.433735 |  |  |  | 85 |
| 4/28/2019 | 349 | D\_N | 1.075269 |  |  |  | 83 |
| 4/29/2019 | 375 | D\_N | 1.052632 |  |  |  | 81 |
| 4/30/2019 | 404 | D\_N | 1.030928 |  |  |  | 79 |
| 5/13/2019 | 658 | D\_N | 9.009009 | 0.884955752 | 11 | 1 | 77 |

Table : Index D\_N

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 ballpark and the fill factor is not 100 as it started out to be. Looking at the graph for this index we get

Figure : Index D\_N

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 well. 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 screenshots from the free version of DPA (Solarwinds Database Performance Analyzer). The first was taken late Feb 2019:

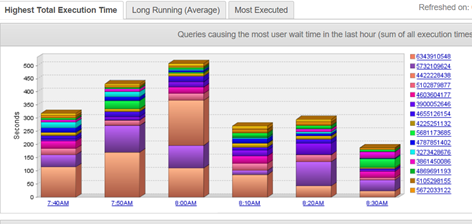


Figure 5: DPA Screenshot from late Feb 2019

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

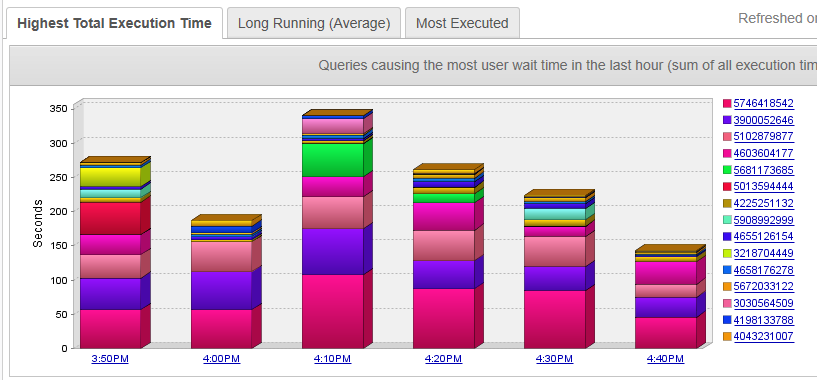


Figure : DPA Screenshot from late June 2019

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 Screenshots was 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 addressing are:

* Add code for multiple databases on the 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 identify “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 wishlist ☺} ).
* Try and build in additional features as suggested by the community
* More testing on performance improvements with firm numbers.
* Need a better way of monitoring Admin.AgentIndexRebuilds – both manually and also self-reviewing. Am looking for a volunteer to implement a dashboard.
* 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 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.

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