Optimizing Fill-factors for SQL Server

Microsoft Corporation
Published: May 2010

Author: Ken Lassesen
Reviewers: Richard Waymire (Solid Quality Mentors),

Abstract

This white paper provides best practices for configuring fill-factors on SQL Server databases. Implementing these best practices can help you avoid or minimize common problems and optimize the performance of SQL Server so that you can effectively manage your resources, reduce operating expenses, increase productivity, and improve employee satisfaction.

Microsoft

Copyright Information

About the Author

Ken Lassesen has been involved with SQL Server performance testing since the first betas of SQL Server. Ken has a Master of Science in Commerce with a focus on Operations Research, Information Management and Statistics. Microsoft's Internal Technology Group gave him the nickname of "Dr. Science" because of the skills he brought to both building a stressing toolkit and analyzing the results for some of the world's largest SQL Server databases. Ken is also known as being part of the original Dr.GUI team for MSDN.

Ken is semi-retired and enjoys frequent hikes in the North Cascade Mountains.

Table of Contents

TABLES

FIGURES

TSQL

Overview

Common practices with SQL Server applications are to use the default fill-factor of 0 (which means 100%) or with the fill-factor that the application uses at installation. This white paper explains why fill-factors need continuously tuning for maximum performance from SQL Server. There are two optimal values possible:

- The maximum throughput fill-factor: a value resulting in the smallest total read-time for indexes between index defragmentation.
- The threshold performance fill-factor: a value resulting in the smallest single read-time for indexes between index defragmentation.

Fill-factors vary according to the index key size and the growth percentage expected between index defragmentation. Optimal fill-factor varies between indexes.

This white paper will illustrate how fill-factors impacts index performance. TSQL functions are provided that will allow you to programmatically adjust fill-factor on your indexes.

The analysis in this paper simplifies down some of the technical details without invalidating the conclusions. The number of records added in any period varies and thus has the optimal fill-factor is a prediction with some variance.

What is a Fill-factor

The fill-factor option is available on the CREATE INDEX and ALTER INDEX statements and provide for fine-tuning index data storage and performance. The fill-factor value affects an index when it is created or rebuilt. The fill-factor determines the percentage of space on each page filled with data. The unused space is reserved for future growth. For example, specifying a fill-factor value of 90 means that 10 percent of each page will empty (excluding the root page). As data fills the table, the index keys use this space. The fill-factor value is an *integer* from 0 to 100 representing the percentage. The server-wide default is 0 which is deemed to be 100.

For clarity, we will use the following terms:

- ***Key entry*** – this is the value of the key, it is also called a filled index row.
- ***Empty slot*** – an empty index row.
- ***Page*** – this is the smallest physical container for a group of keys, there are 8 pages in an extent. Extents can be mixed, for example, one extent may contain eight different small indexes.
- ***Defragmentation*** – this may occur because of either a reorganization or rebuild of indexes.

Current state of the literature

If you open up many SQL Server Reference books and look at fill-factor in the book's index, you will find typically only one or two pages dealing with fill-factor. Often any advice is fuzzy and do not answer the question "What number should I set the fill-factor to?" The quote below illustrates this.

Best Practice

The best fill-factor depends on the purpose of the database and the type of clustered index If the database is primarily for data retrieval, or the primary entry is sequential, a high fill-factor will pack as much as possible in an index page. If the clusterered index is nonsequential (such as a natural primary entry), then the table is susceptible to page splits, so use a lower page fill-factor and defragment the pages often.[1]

What could be the impact of Fill-factor on Performance

This white paper arose out of scratching out on a napkin some numbers on what you would expect for the time to read a full index with and without fragmentation. Using published latency figures for a SCSI 15000 RPM hard drive we discover that 0.5 milliseconds is the time to go from one track to the adjacent track. If we are jumping to another track location, the time averages 3.5 milliseconds with up to 8 milliseconds being possible. If we have 99 adjacent tracks and 1 fragmented track somewhere in the index, we have 98 adjacent reads @ 0.5 milliseconds, with one head movement to the fragmented track and then a head movement back from the fragmented track (2 @ 3.5) for a total of 56 milliseconds compared to 50 milliseconds with no fragmentation. ***That is a 12% increase of read time for a 1% fragmentation***. To make matters worse, a 1% fragmentation could occur with less than 1% growth of the table.

If we ignore a host of factors that might influence performance (such as the index already being in cache) and focus on the physical read we see some interesting facts when we look at an index with 100 pages and various degrees of fragmentation as shown in the Table 1 below. Physical records are read either *adjacent to* the prior record read, or *disjoint from* the prior record. A disjoint record means that the physical head on the disk spindles must move (3.5+ milliseconds). An adjacent record means that the physical head is already at the right location (0.5 milliseconds).

To calculate the time for 5 fragmented pages in 100 pages, we have 100-5= 95 times that we have adjacent reads (90 x 0.5 = 47.5 milliseconds), and 5 times that we have 5 disjoint reads (5 x 7 milliseconds = 35 milliseconds [3.5 milliseconds to move there and 3.5 milliseconds to move back]) for a total of 82.5 milliseconds. The increase in time is (82.5-50)/50 = 65% with this 5% fragmentation.

[1] P. 554, **Microsoft SQL Server 2008 Bible**, (Wiley Publishing, 2009), Paul Nielsen, Mike White, Uttam Parui.

Table 1 -- Example of physical read times of an Index on 100 pages with fragmentation

Fragmented Page Count	Adjacent Read Time	Disjoint Read	Total Time	Increased Read Time
0	50.0	0	50.0	0
1	49.5	7	56.5	13%
2	49.0	14	63.0	26%
3	48.5	21	69.5	39%
4	48.0	28	76.0	52%
5	47.5	35	82.5	65%
6	47.0	42	89.0	78%
7	46.5	49	95.5	91%
8	46.0	56	102.0	104%
9	45.5	63	108.5	117%
10	45.0	70	115.0	130%

The complimentary table, Table 2, shows the time to read with various fill-factors (assuming that we have 100% full pages to start).

Table 2 -- Example of physical read time with different fill-factors and no fragmentation

Fill-factors	Actual Pages	Total Time	Increased Read Time
100%	100	50.0	0%
99%	102	51.0	2%
98%	103	51.5	3%
97%	104	52.0	4%
96%	105	52.5	5%
95%	106	53.0	6%
94%	107	53.5	7%
93%	108	54.0	8%
92%	109	54.5	9%
91%	110	55.0	10%
90%	112	56.0	12%
89%	113	56.5	13%
88%	114	57.0	14%
87%	115	57.5	15%
86%	117	58.5	17%
85%	118	59.0	18%
84%	120	60.0	20%
83%	121	60.5	21%

Assume you rebuild the indexes once a week, which is better?

- 95% Fill-factor with 7% fragmented records before the rebuild, or
- 83% Fill-factor with 1% fragmented record before the rebuild

Using the above tables, we find an expected 47% increased read time for the first choice and an expected 26% for the second choice. The approximate calculation to determine average read time is:

{Fill-factor % Increased} + {Fragmented % Read Time}/2

(The fragmentation will be zero at the start and increase to the end value, thus having an average of ½ the end value.)

What we have is a mathematics problem of ***finding the fill-factor that result in the least expected increase of read time,*** **the optimal fill-factor*.*** The mathematics gets a little complex because the fill-factor influences the fragmentation. A 100% fill-factor will cause a fragmentation on the first new record. A 50% fill-factor will not fragment on the first new record.

Another bit of the mathematics complexity is that we are dealing with integer values. You cannot have 101.5 pages; it is actually 101 or 102 pages. You cannot have a fill-factor of 87.5%, you can only specify 87% or 88%. When we look at the number of key entries in a page, we find that the entry key size divided into 8060 bytes is the maximum number of values in a page. If the entry length is 900 bytes, then 8060/900= 8.955, which means that there is only eight key entries in a page. We must always round downwards because we cannot create 0.955 of a value. In this case, with a key entry length of 900 we can only have eight records in a page so there is no difference between a fill-factor of 99% or 88%. Both results in one empty slot with the actual physical fill-factor being 87.5% (7/8) which could be entered as any value from 88% to 99%. Entering a value of 87% will result in 2 empty slots and 6 fixed slots.

The computation of optimal fill factor is unfamiliar to most SQL Server developers, architects and administrators. This paper covers both complex theory and practical implementation. An architect may be just interested in the following sections:

- Indexes in SQL Server
- Operational Use
- Summary

A reluctant DBA may wish to skip directly to the code in

Appendix C: Automatic One Time Tuning of Fill-Factor and execute it. A SQL Server performance analyst may wish to read every section.

Quick Acid Test

"Do I need to care about tuning fill-factor for indexes, after all, I'm sure that the developers have already done this!" Common practice in the industry is to go with the default value of 0 which is the *best fill-factor* for a clustered index on an identity column. For any other type of key, it produces the worst performance. The value of 0 is an artifact from the early days of SQL Server when most tables used identity columns as a primary key – this was the right decision for those days. Today indexes often include unique identifiers (GUIDs) to support replication and other recent feature additions.

The TSQL below gives a distribution of fill factors. Typically, you will see every index having a value of 0.

TSQL 1 -- Retrieve Count of Fill-Factor Used in a Database

```
select fill_factor, count(1) as NoOfIndexes
from sys.indexes
group by fill_factor
```

A more refine distribution adds the key size of each index to the results as shown in the TSQL below.

TSQL 2 -- Retrieve Count of Fill-Factor by Key Size in a Database

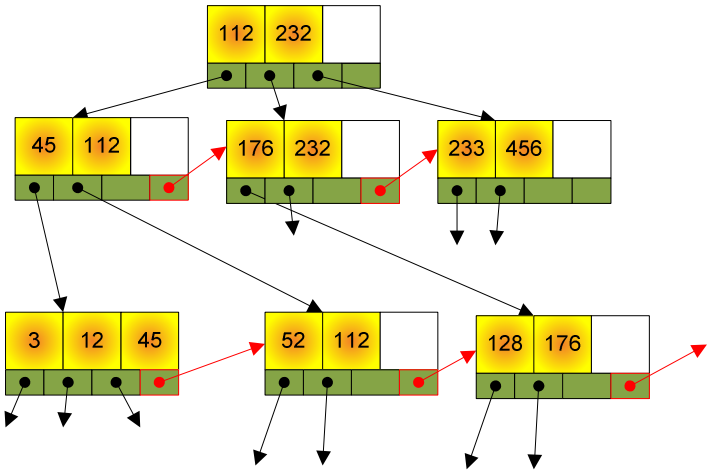
```
Select
      [KeySize],
      [Fill_Factor],
      Count(*) As [NoOfIndexes]
FROM (Select
            [database_id],
            [object_id],
            [index_id],
            max(max_record_size_in_Bytes) as [KeySize]
      FROM sys.dm_db_index_physical_stats(
    db_id(), NULL, NULL, NULL, 'DETAILED')
    Where index_type_desc IN ('CLUSTERED INDEX','NONCLUSTERED INDEX')
      AND alloc_unit_type_desc='IN_ROW_DATA'
      Group by [database_id],[object_id],[index_id]
      HAVING Sum(page_count) > 0
) S
Join Sys.objects T
      On S.[object_id] = T.[object_id]
Join Sys.schemas sch
      On Sch.[schema_id] = T.[schema_id]
Join Sys.Indexes
      On Indexes.[Object_id] = S.[Object_Id]
      And Indexes.[Index_id] = S.[Index_id]
Group by S.KeySize, [Fill_Factor]
```

Indexes in SQL Server

SQL Server indexes are a variation of balanced tree structures (B-tree) called B+ trees. Key entries are limited to a maximum size of 900 bytes (8 key entries per page) and not more than 16 columns in the key. Usage of B+ is common with operating systems and databases dating back to IBM VSAM in 1973.

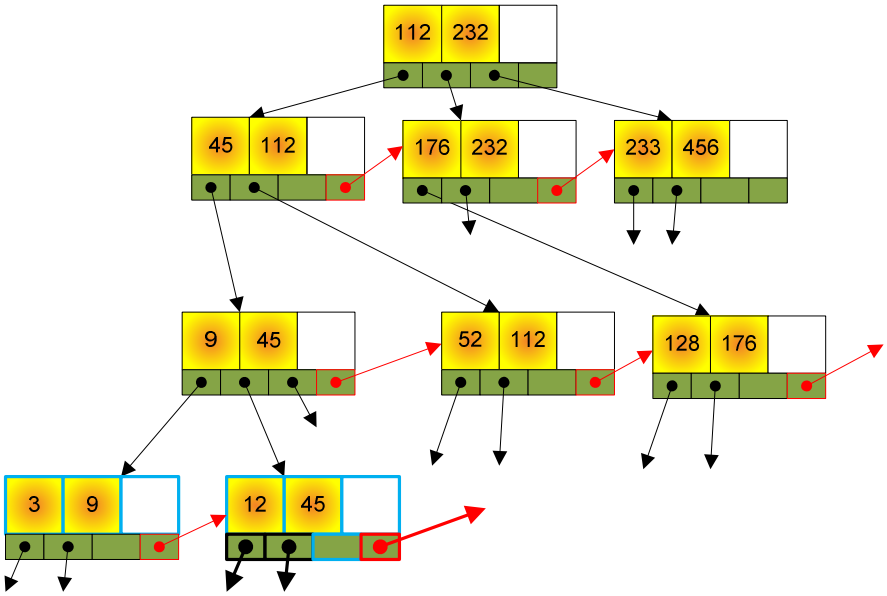
The simple visualization shown below in Figure 1 assuming three key entries per page (the minimum number of key entries with SQL Server is eight). The blank squares are free space for future key entries. The linked list pointers are red arrows.

Figure 1 -- Visual Example of B+ Tree



If we insert a new entry [9] into the B+ tree, a split might occur as shown in Figure 2. The location of the two new pages may be at a considerable distance from the neighboring pages resulting in the hard drive heads having to do an expensive seek operation.

Figure 2 -- Example of a split



When a page splits, it goes from 100% full to two pages that are *at least 50%* full. If you had 9 key entries in a single page before the addition of a entry (bringing it to 10 key entries), the two new pages contains 5 filled key entries (55% filled) and 4 unfilled key entries in each of the resulting pages. With 10 key entries in a single page before the addition of a entry (bringing it to 11 key entries), the two new pages will consisting of:

- one with 5 filled key entries (50% filled) and 5 unfilled key entries in each one resulting pages, and
- 6 filled key entries (60% filled) and 4 unfilled key entries in the other resulting page.

A page spilt will never have less than 50% filled.
To see information about your indexes try the query below that references [sys.dm_db_index_physical_stats]

TSQL 3 -- Retrieve Fragmentation of Indexes

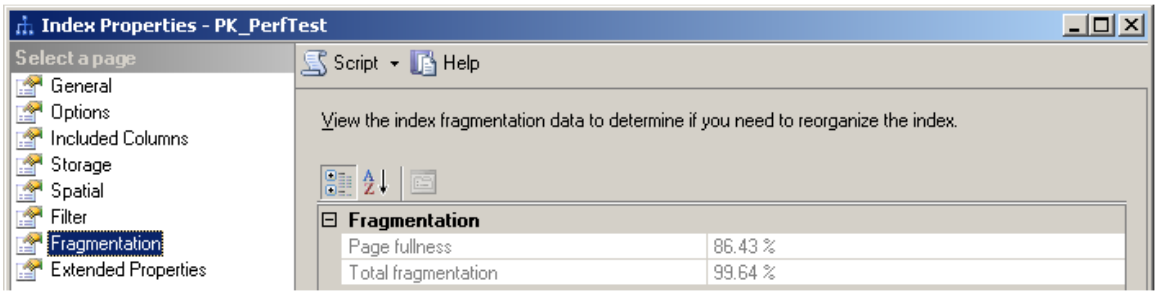
```
SELECT
       object_id,
       index_level,
       avg_fragmentation_in_percent,
       fragment_count,
       page_count,
       avg_page_space_used_in_percent,
       record_count
FROM sys.dm_db_index_physical_stats(
     db_id(), NULL, NULL, NULL, 'DETAILED')
     Where index_type_desc <> 'HEAP'
     AND alloc_unit_type_desc='IN_ROW_DATA'
```

Figure 3 shows a three level index with a very high fragmentation rate at one level (99.64%).

Figure 3 -- Example of a fragmented index

Results | Messages

	object_id	index_level	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count
1	180911716	0	99.6390615600561	9974	9974	86.434815913022	428974
2	180911716	1	62.962962962963	27	27	59.3068569310601	9974
3	180911716	2	0	1	1	4.31183592784779	27



For further information about B+ tree behavior see:

- B-Trees: Balanced Tree Data Structures
- COS 597A: Principles of Database and Information Systems B+-tree insert and delete, Princeton, 2008, by Andrea LaPaugh
- **Microsoft SQL Server 2008 Internals**, Kalen Delaney, Microsoft Press, 2009, pp 299-374.

Technical Analysis

The technical analysis focuses on determining the optimal fill-factor. Looking at a B+tree we see many behaviors: reading across an index with the linked list pointers (the red arrows in Figure 1), reading down the tree, and many variations[2]. Fortunately, optimizing the performance for some of these behaviors, results in optimizing for the others -- we will use the reading across an index to do our analysis because it is a simpler model.

A second aspect is the caching behavior of SQL Server and hardware. This becomes complex and depends on whether there are sufficient memory to keep all indexes cached in memory, etc. When SQL Server is starting up, and when SQL Server is running with limited memory, then reading from disk will happen. The operating system and hardware (controller, drives) caching influence performance. For simplifying the analysis to find the optimal fill-factor, we will assume no caching occurs anywhere. When caching occurs, it may reduce the difference by a significant percentage or to zero. Ignoring caching does not change the optimal fill-factor; it only changes the amount of performance improvement expected.

Caveat: This analysis ignores some issues that would complicate the analysis but typically have low significance to the conclusions. The following index types are not examined: PRIMARY XML INDEX, SPATIAL INDEX, XML INDEX.

For purposes of analysis, we will assume the following values:

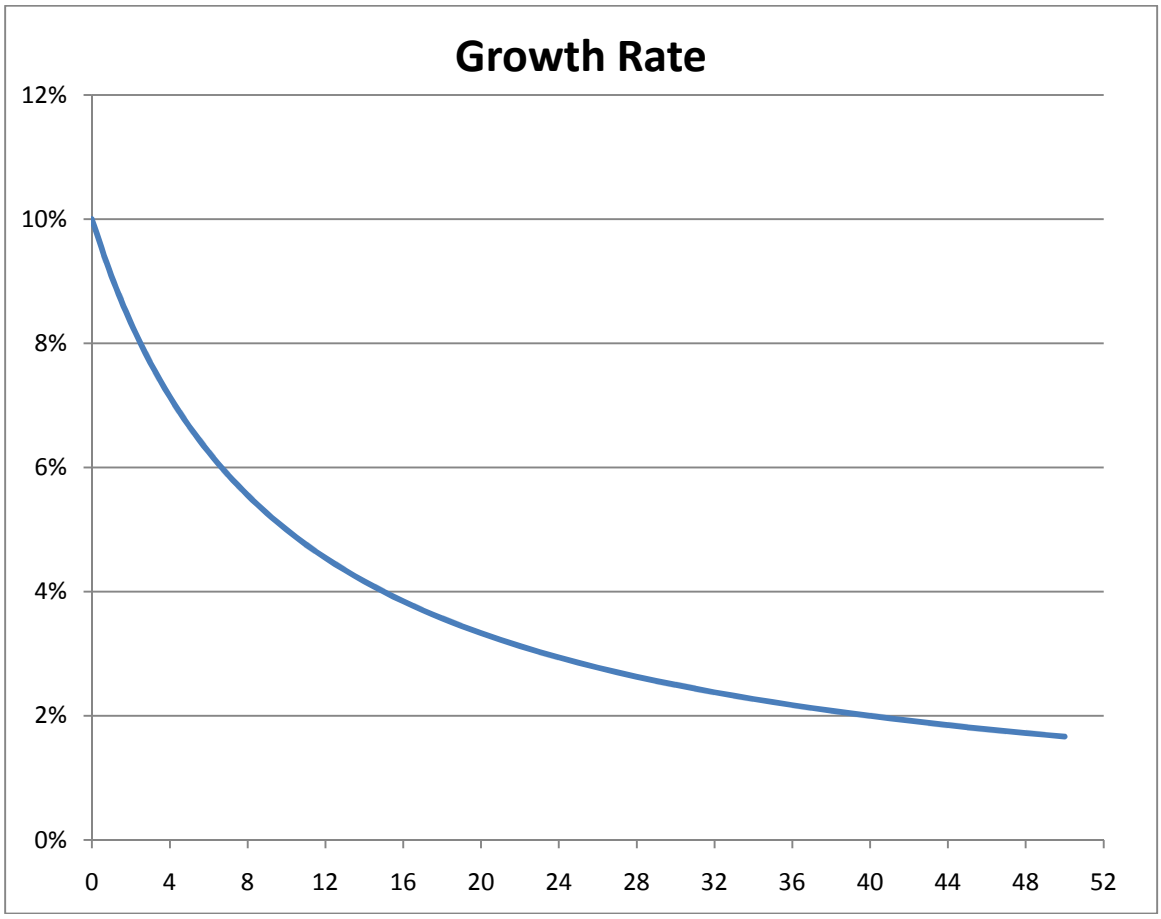
- 0.5 milliseconds to move from one page to the next if there is no fragmentation
- 4.5 milliseconds to move from one page to a fragmented page, followed by a 4.5 milliseconds to return to the original pages disk area.

These numbers are compatible with the behavior seen on contemporary 15000 RPM SCSI drives.

Most DBAs schedule defragmentation of indexes on a periodic basis. The dynamics of fragmentation changes over time because the index relative growth rate keeps declining over time. In Figure 4, we see how *with a constant rate of new records*, the percentage of new records decrease. If the index defragmentation occurs at the end of each period, we see that the maximum fragmentation possible at the end of 52 periods is less than 2% with appropriate fill-factor. Our analysis base is the percentage growth between defragmentation. Since percentage growth changes (typically monotonic decreasing) over time, the optimal fill-factor *will be dynamic* and a function of the growth rate.

[2] See “Index Access Methods”, pp 582- 593, **Professional SQL Server 2008 Administration** (Wiley, 2009), Brian Knight, Katen Parel, Steven Wort, et al.

Figure 4 -- Weekly growth rate with constant period inserts



The dimensions of optimizing fill-factor for an index summarizes in Figure 5:

Figure 5 -- Fill-factor Function showing dimensions

Fill-factor = Function (Key Size, Growth Rate between Index defragmentation, Index Type)

The following are not critical factors:

- Seek times between adjacent pages and fragmented pages
    - Assuming the first is less than the second
    - The magnitude of performance differences is influenced
- Time between index defragmentation
    - Influences the growth rate which is already captured above
- Number of Pages in an Index
    - The empty slots are in each page, so the number of pages will change because of the fill-factor.

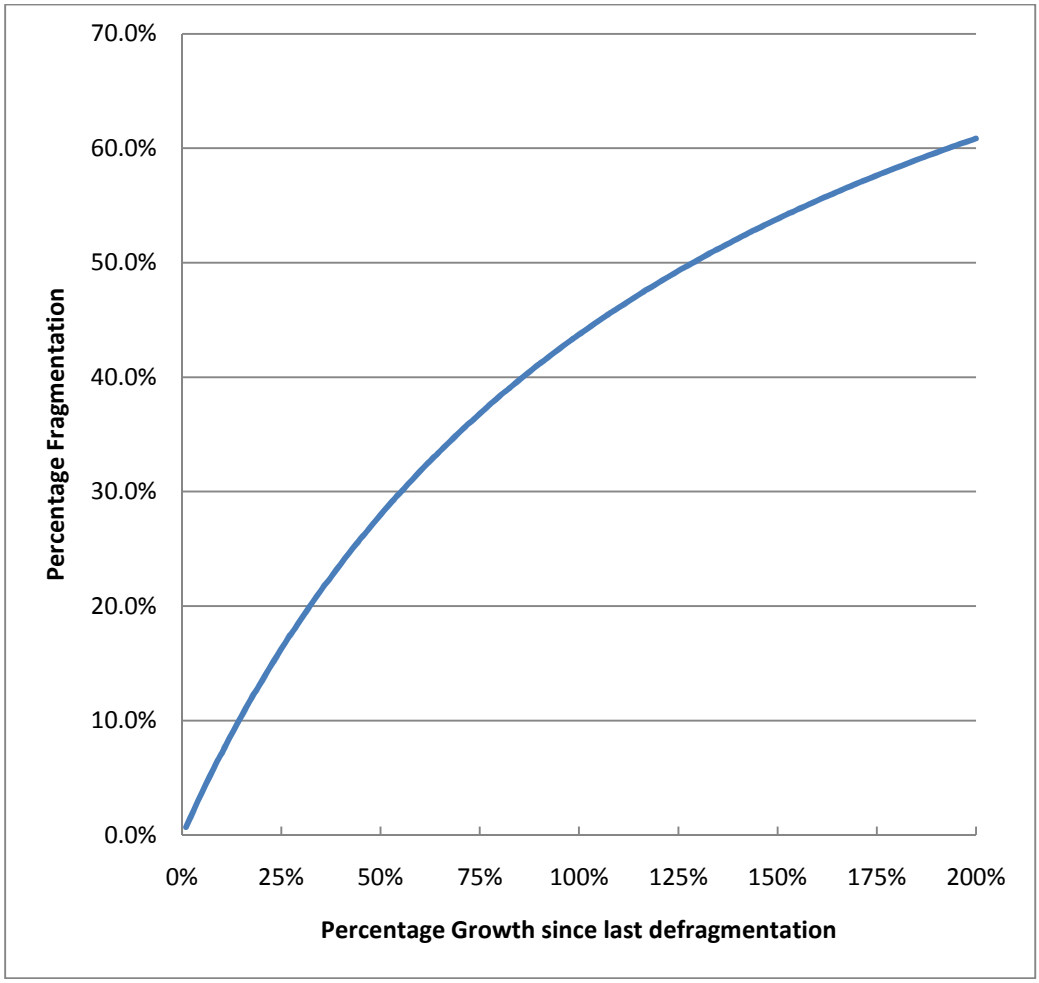
Index types

SQL Server has many types of indexes. Clustered indexes and non-clustered index are the classic ones. For fill-factor, the critical item is whether new key entries are monotonic increasing or arbitrary. The easiest to understand is a unique clustered index based on an Identity.

Monotonic Increasing Key entries

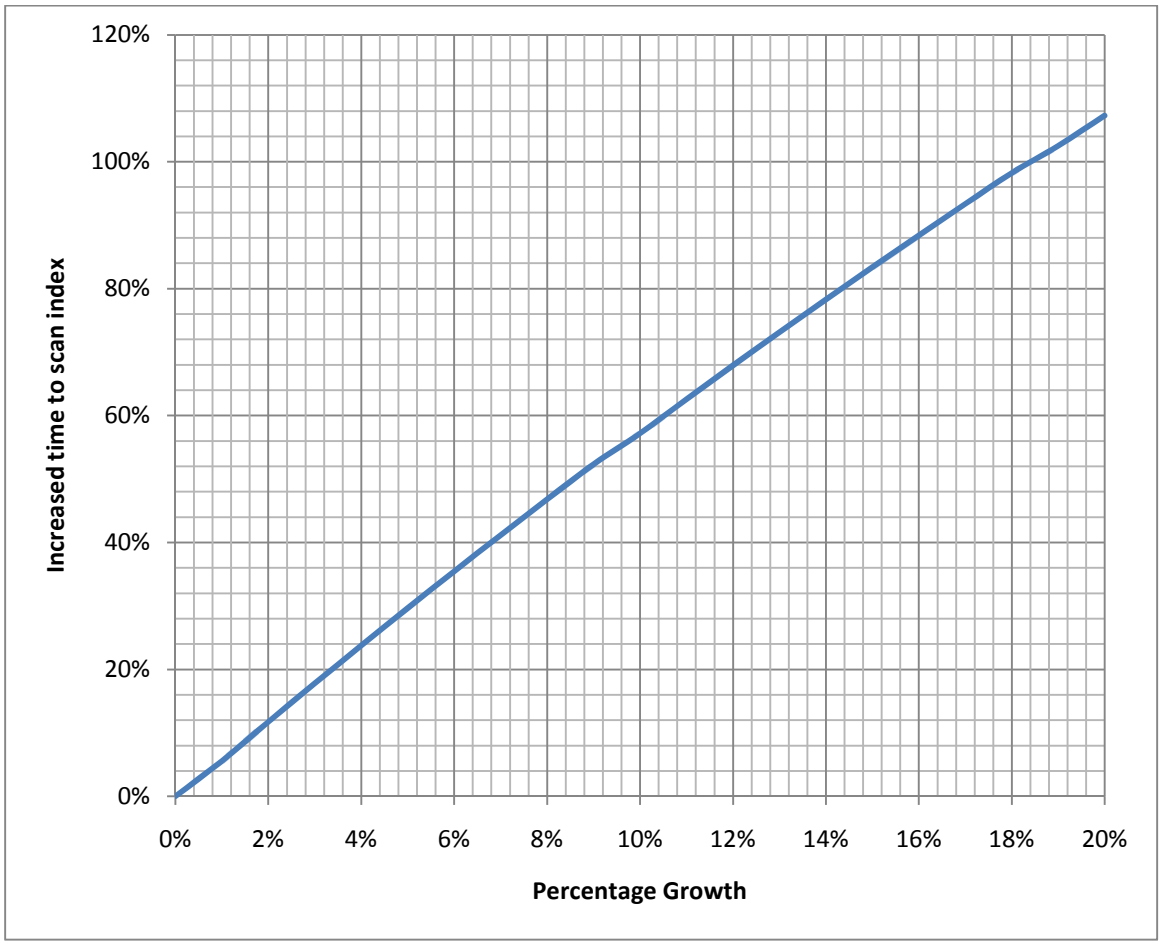
An index with monotonic increasing key entries will always have new key entries added at the end of the index. Any empty entry slots left in a page is never used and will slow down the read of the index because they will be more pages to read. The fill-factor should always be 100% (or the default fill-factor of 0 for SQL Server). The fragmentation rate versus growth rate is shown in Figure 6. Growth impact on Index read time is shown in Figure 7.

Figure 6 -- Fragmentation Rate with Monotonic Increasing Keys and 100% Fill-factor



A 2% growth results in a 11.7% increase of index read time as shown in Figure 7.

Figure 7 -- Increased time to read index due to fragmentation with Monotonic Increasing Keys and 100% Fill-factor



Typical monotonic increasing key entries include:

- Identity on any integer data type
- timestamp
- rowversion
- datetime, datetime2 or datetimeoffset defaulted to GetDate() or GetUtcDate() and *never* assigned by code

Arbitrary Key entries

Arbitrary key entries means that key entries that are not monotonic increasing key entries. Key entries are inserted into multiple pages and might result in splits / fragmentation. The mathematics becomes complicated because we are dealing with a discipline called Integer Linear Programming. Our analysis will focused on random keys, for example the keys produced by using NewId() which produces a GUID.

A walk thru will illustrate the process.

Consider a key entry size of 900 bytes. This results in 8060/900 → 8 key entries per page. This means that there are only eight possible values for fill-factor (actual values should be rounded up is a decimal is shown).

- 100% - 8 key entries per page
- 87.5% - 7 key entries per page
- 75% - 6 key entries per page
- 62.5% - 5 key entries per page
- 50% - 4 key entries per page
- 37.5% - 3 key entries per page
- 25% - 2 key entries per page
- 12.5% - 1 entry per page (the minimum allowed)

Any other value will result in the fill-factor *rounding down* to one of these values. The remaining variable in Figure 5 is percentage growth.

We can easily determine boundary values by considering a simple deterministic model: what happens when we have just one page that we know will be filled before we defragment the index, as shown in Table 3. To determine the fill-factor just scan down the growth percentage until you exceed or equal your expected growth percentage.

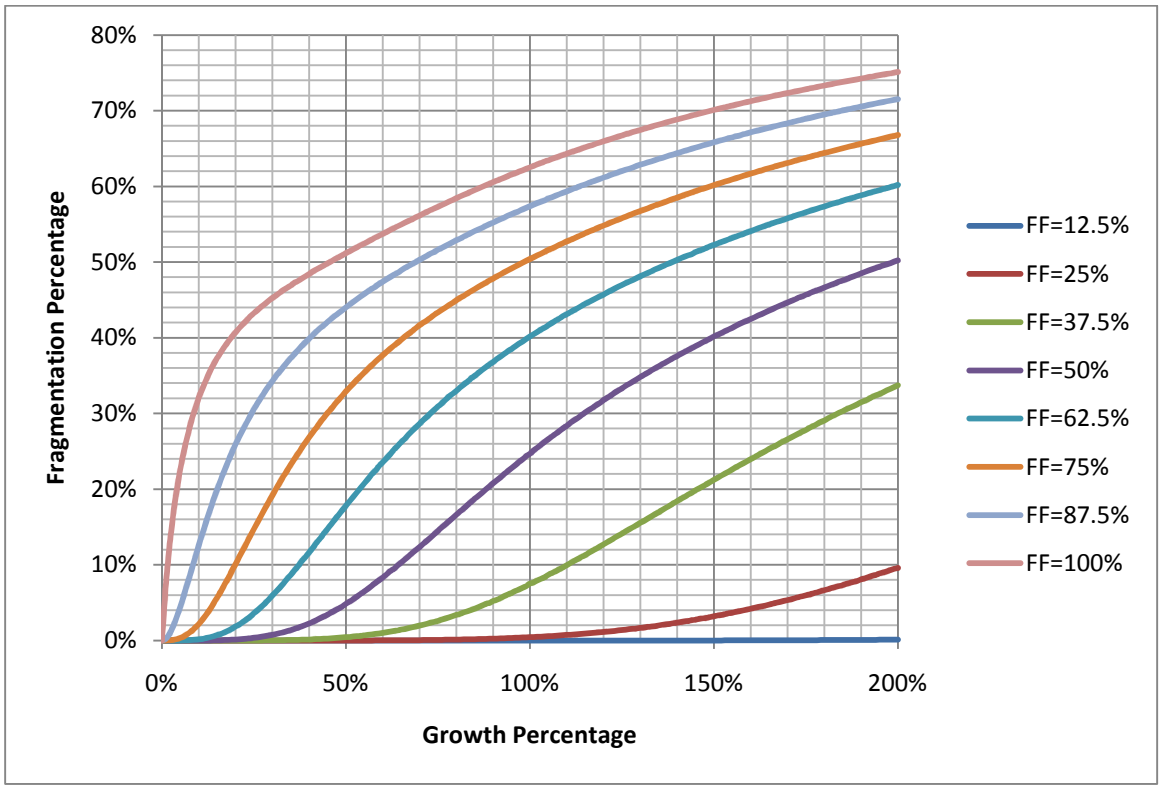
Table 3 -- Simplistic Fill-factor for Largest Key Size

Fill-factor	Existing Key Entries	Slots to Fill	Growth Percentage
87.5%	7	1	14%
75%	6	2	33%
62.5%	5	3	60%
50%	4	4	100%
37.5%	3	5	166%
25%	2	6	300%
12.5%	1	7	700%

Unfortunately, we are not able to say deterministically that no more than four key entries occur when we have four filled slots. When we move to realistic scenarios, we might encounter thousands of pages and have some probability distribution on the number of records expected compounded by some distribution of where the new key entries locations. To deal with these complexities, we will assume the normality (normal distribution).

Assuming a random distribution of new records, we see how the fill-factor influences the fragmentation percentage for our 900 bytes key example in Figure 8. A typical index that is random is an index with its first column being a unique identifier (GUID).

Figure 8 -- Fragmentation Percentage resulting from different fill-factors



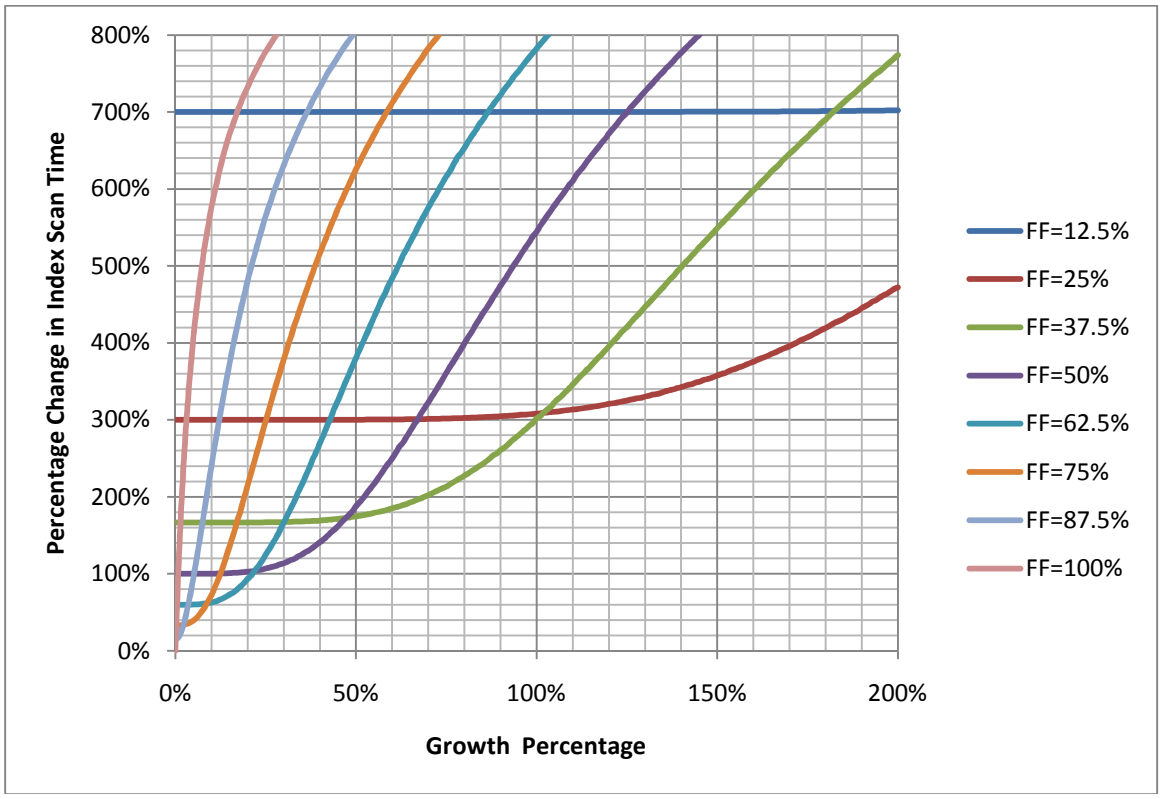
Consider a fill-factor with 25% (FF=25%), that means that we have 6 empty slots. With a growth rate of 200%, it means that the average page will move from 2 of 8 slots filled to 4 of 8 slots filled. About 10% of the extents will receive more than 6 new key entries due to randomness and thus split, resulting in 10% fragmentation.

Our concern is the time to scan the entire index and not the fragmentation percentage. These two measures are related, but not linearly. Consider the case of FF=12.5%, we have almost zero fragmentation but the size of the index is 8 times the size of an index with a FF=100%, and takes 8 times as long to read. The next chart, Figure 9, shows the time to read the index (in terms of a not-fragmented fill-factor=100% index of the same size).

Our goal is to have the lowest read time, so to use this chart for 900 byte keys, we simply find the percentage that we expect to grow and then go up until we hit a line. The first line is the optimal fill percentage. Some examples:

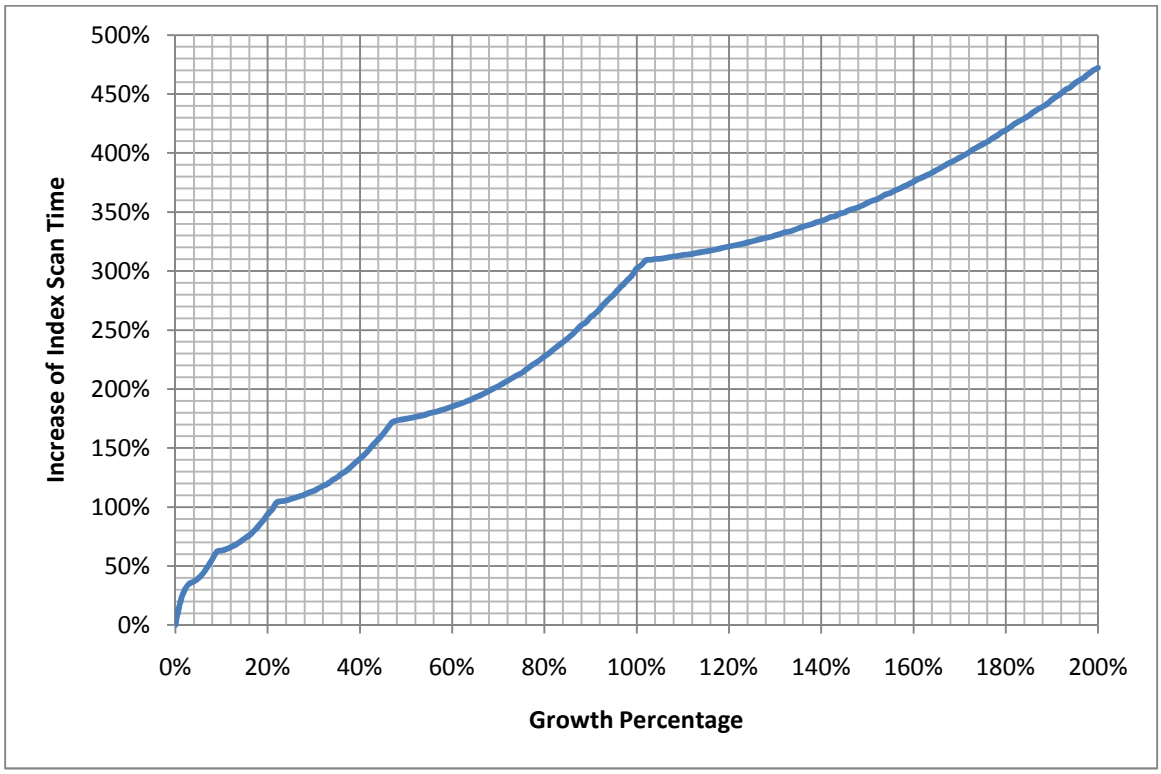
- At 200% expected growth, a fill-factor of 25%
- At 100% expected growth, a fill-factor of 37.5%
- At 40% expected growth, a fill-factor of 50%
- At 8% expected growth, a fill-factor of 75%

Figure 9 -- Increase of Index Read Time versus Growth for different Fill-factors



We can extract from the above curve of optimal values, as shown in Figure 10.

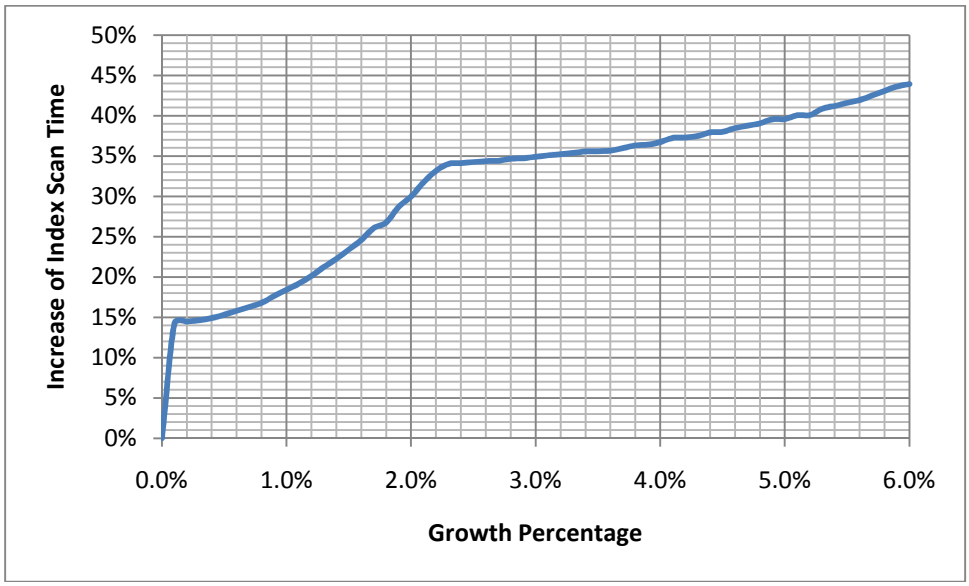
Figure 10 -- Extracted Chart of performance with optimal values



In reality, with weekly or daily index defragmentation, the growth percentage will be 6% or less, so we extracted into Figure 11 that portion of the data. We may summarize as follows:

- From 0- 2.2%, a fill-factor of 88%
    - 100% is optimal if less than ~ 0.1%
- From 2.2-6%, a fill-factor of 75%

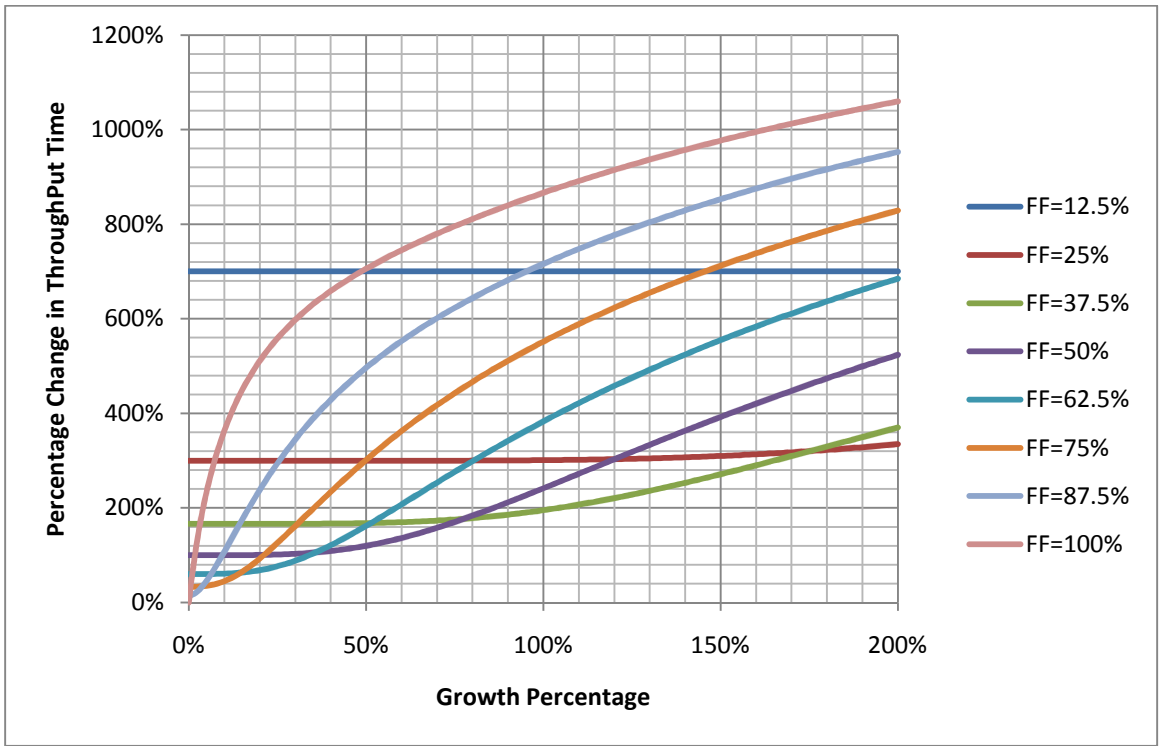
Figure 11 -- Performance with optimal values for 0-6% Growth Factor



The above analysis is correct for the performance at the *end of the period*. This is the **threshold performance fill-factor** – the performance will be at this level or less. This is the equivalent to speed or velocity.

The **maximum throughput fill-factor** is the equivalent of the distance travelled. If we want the best throughput performance over the entire period, we need to use the *integral* for each of the curves shown in Figure 9. Integral means the area under the curve. Charts are shown in Figure 12 and following.

Figure 12 -- Throughput versus Fill-factor versus Growth Percentage



As above, we can extract the optimal performance chart, as shown in Figure 13. Again, we can drill down into the typical 0-6% growth operational range in Figure 14 where we see that the transition point between FF=75% and FF=87.5% is now at 4% and not 2.2%.

Figure 13 -- Optimal Throughput versus Growth Factor

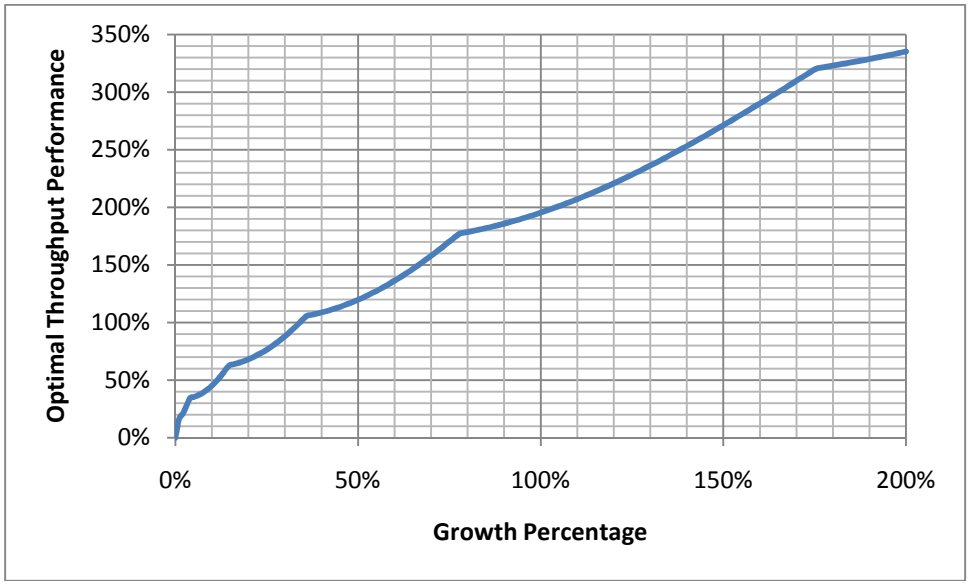
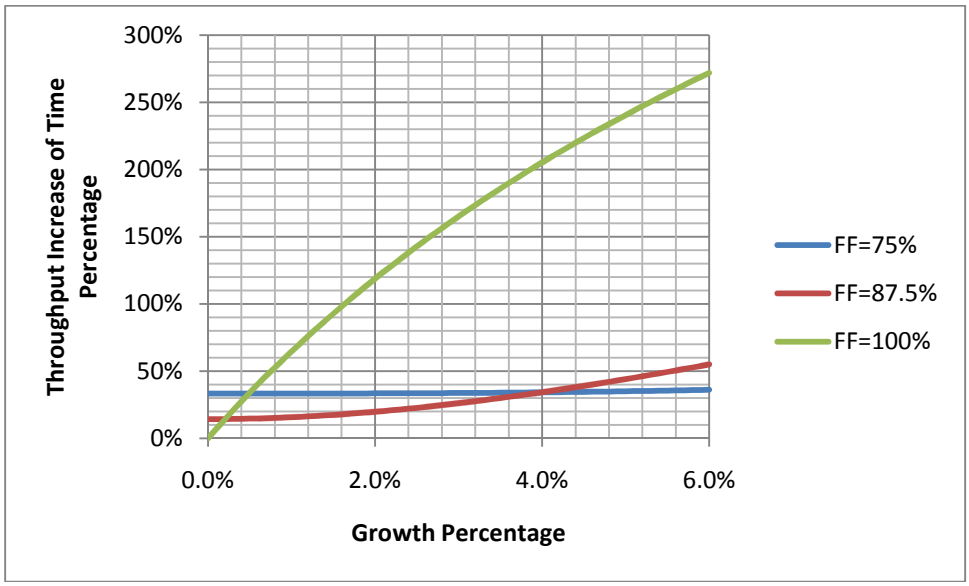


Figure 14 -- Optimal Throughput versus Growth Factor for 0-6%



An additional item that Figure 14 illustrates is how bad a choice the default 100% fill-factor is for performance. A 2% growth results in more than a 125% increase in time.

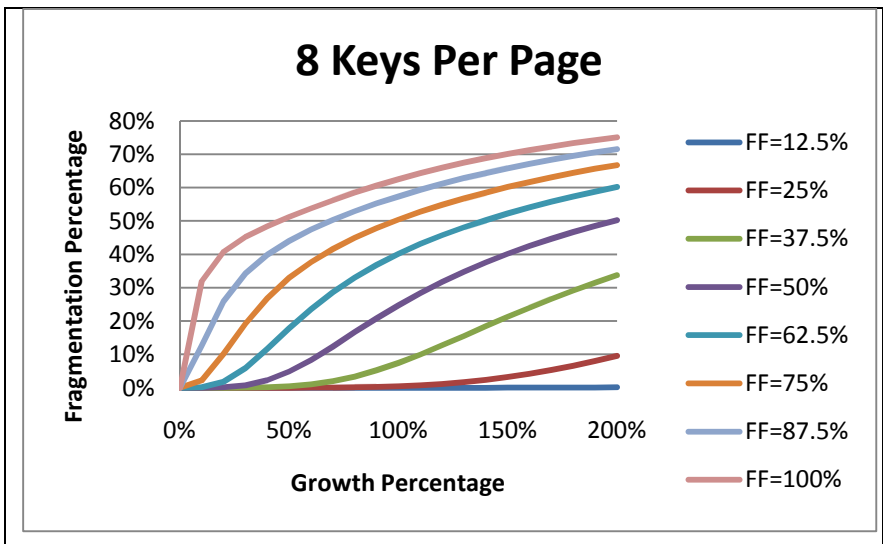
Summary of Examples

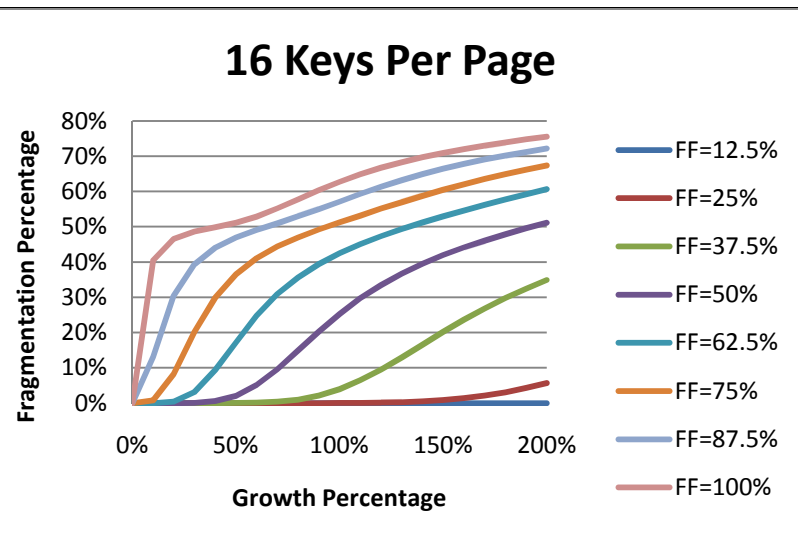
The above charts illustrate how you can find the optimal values visually. The actual mathematics is complex and includes probability, integer programming and integration. All of the above charts were done for a 900 bytes key (8 keys per page). We could repeat the exercise for keys between 810 and 899 bytes (9 keys in a page) and then 736=809 bytes (10 keys in a page), etc.

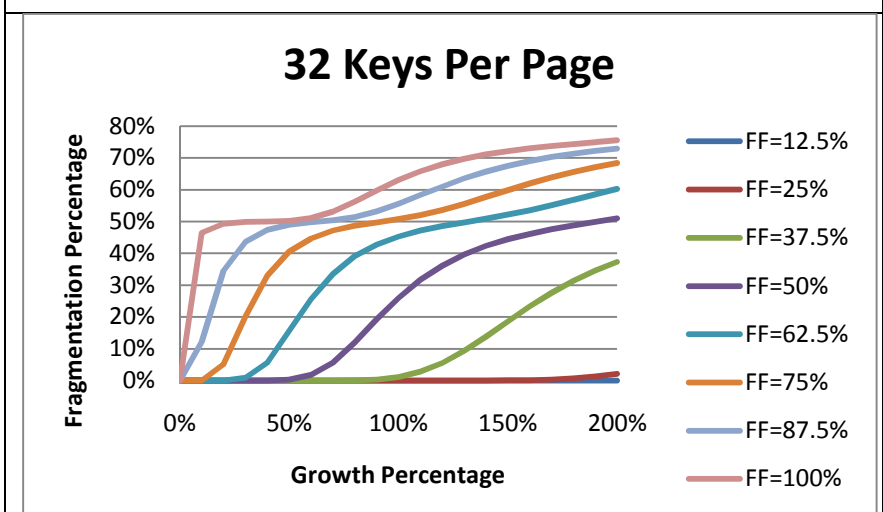
Putting Optimization into Practice

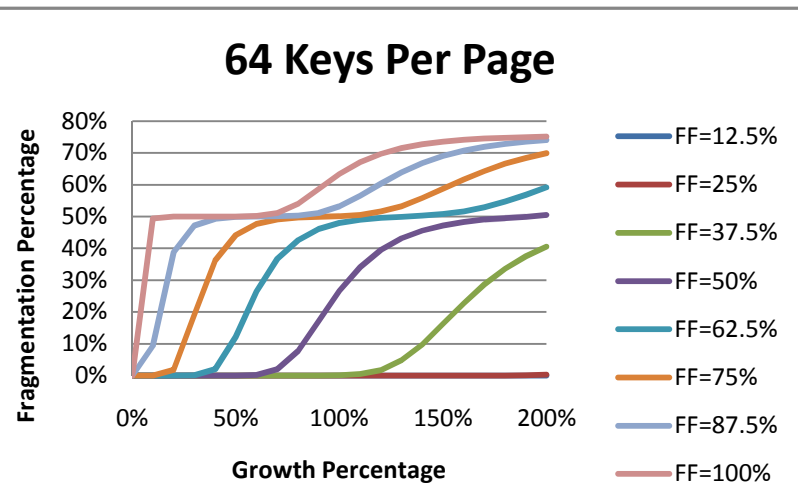
The goal of this white paper was to find the optimal fill-factor given an expected growth rate until the next index defragmentation and a key size in bytes. Above we explored a simple case to illustrate the mechanics involved. Do not jump to the erroneous assumption that the above curves works for all keys. The best way to illustrate this is by looking at Table 4 which contains charts for extents with 8, 16, 32, 64, 128, 256, 512 keys per page, the equivalent of key sizes of 900, 503, 251, 125, 62, 31, 15 (approximately GUID size).

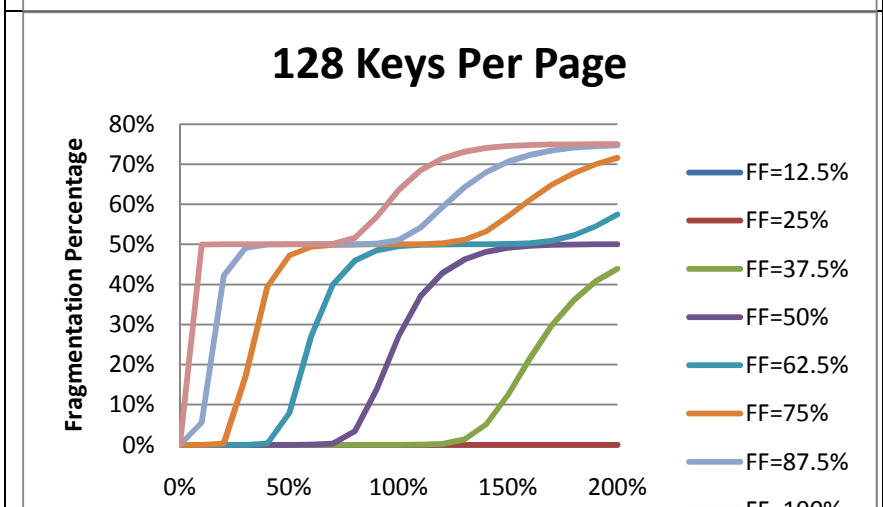
Table 4 -- Fragmentation as a result of keys per page and growth factor

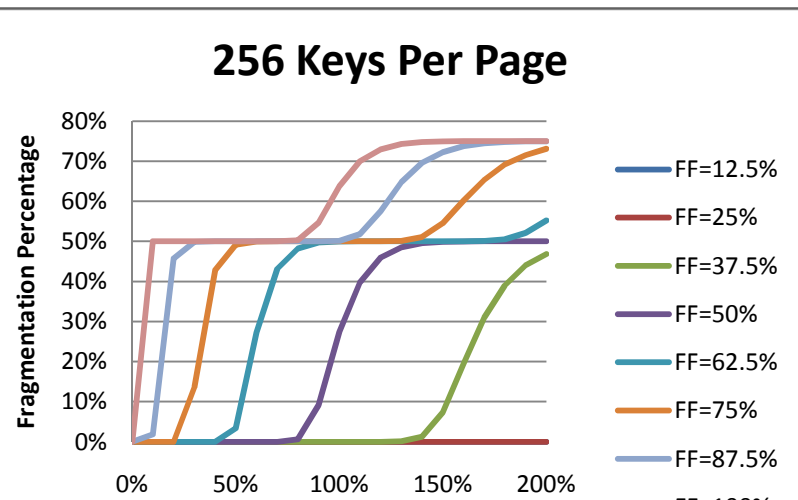


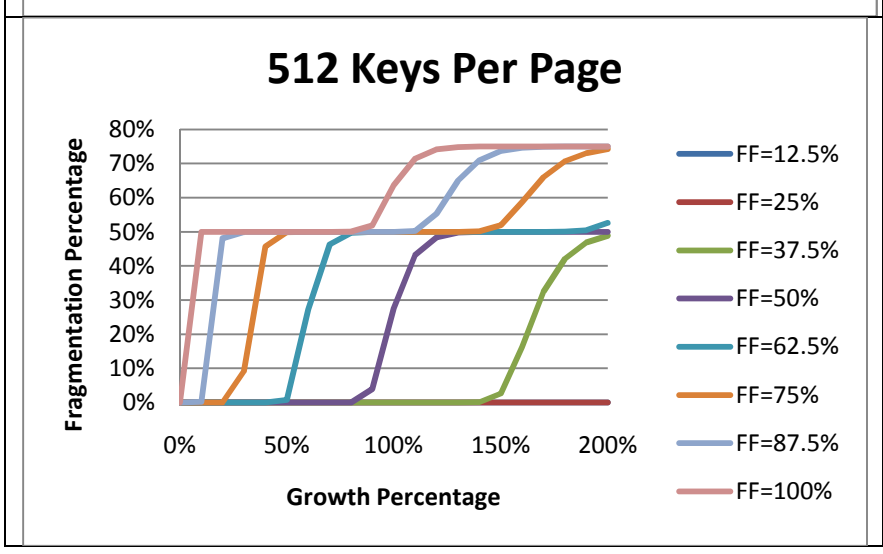












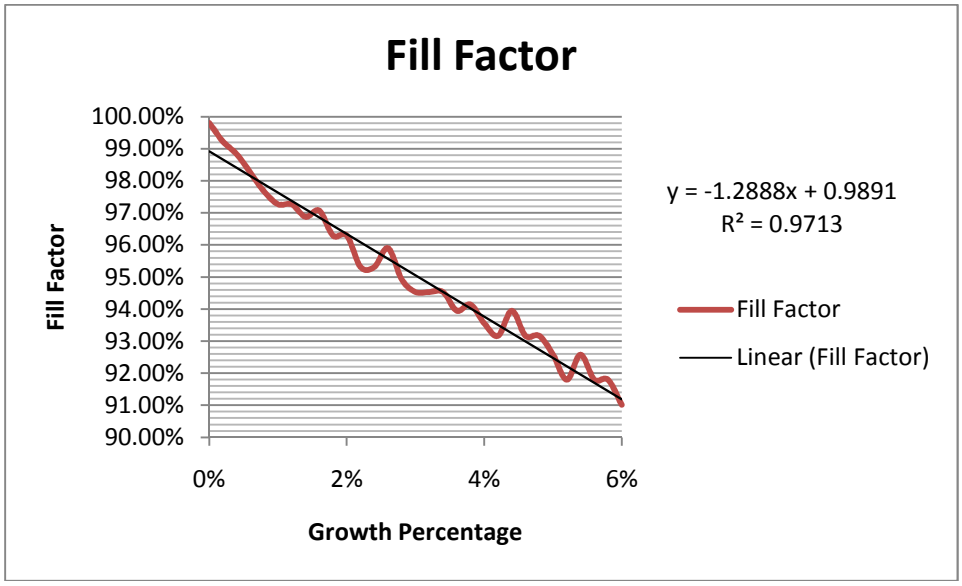
You may wonder why we have a sudden twist in the chart as the key size reduces. The mechanics can be illustrated by considering a 100% fill-factor. Every new key results in a split resulting in two pages that are 50% filled. At the start almost every new record will result in a split because of the 100% fill-factor. Over time the index is dominated with 50% filled pages which slowly filled and then starts a second generation of splits (and another plateau as seen with 512 keys per page).

Operational Range

Finding the optimal fill-factor when more than 6% growth happens is actually counterproductive or extremely atypical. Indexes should be defragmented regularly and a 6% growth should not occur. The above Table 4 illustrates the complexity of the behavior; we will reduce our focus to the operational range of 0-6% growth.

Returning to **threshold performance fill-factor**, we find that there is linear pattern with high correlation for 512 keys in a page which should be sufficient for operational use.

Figure 15 -- Linearity of Optimal Fill-factor and Growth Percentage[3]



Checking the other keys per page we see a similar pattern in Table 5 with correlation decreasing as the number of keys per page decreases.

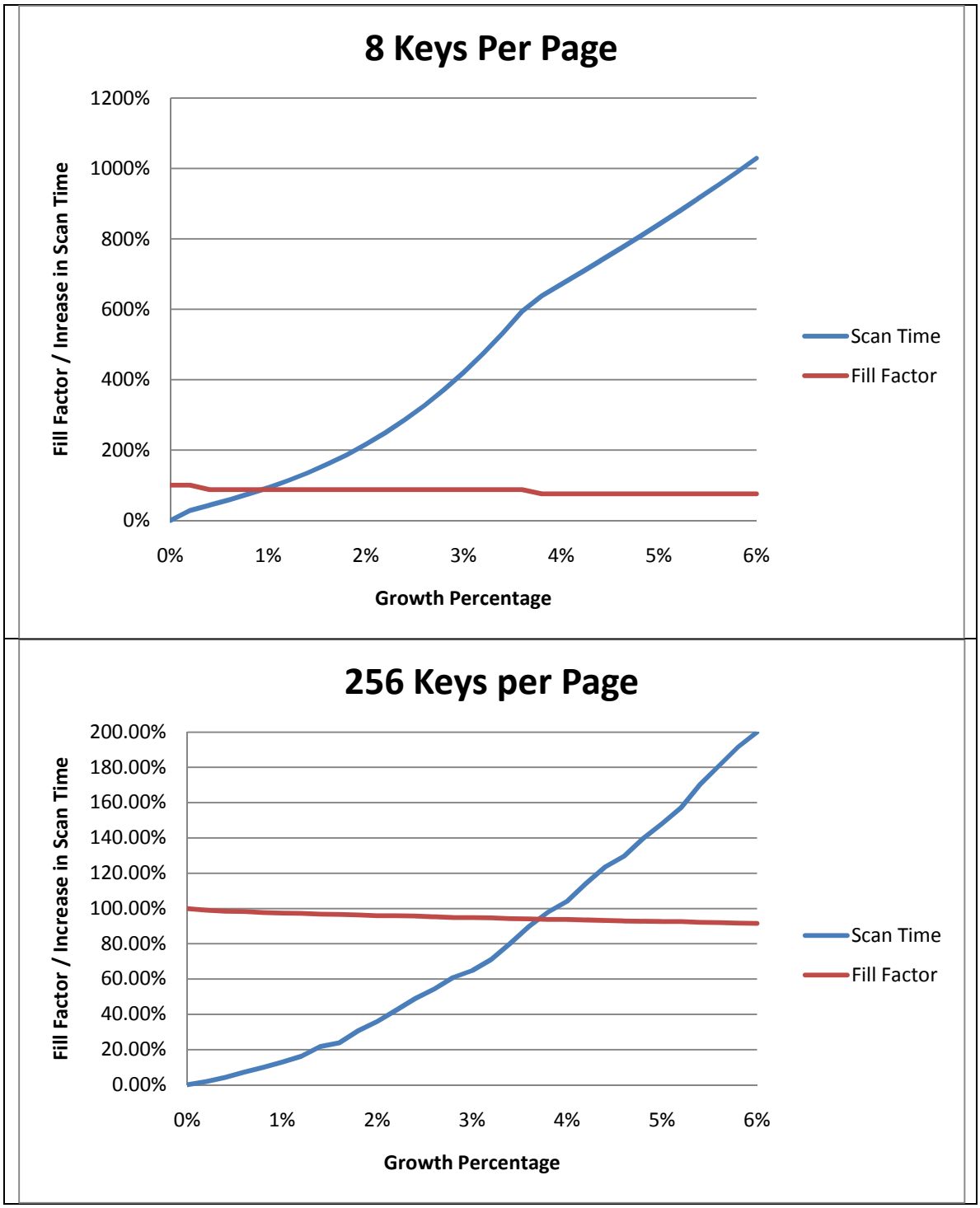
[3] This chart was produced by running 100 simulations for each data point.

Table 5 -- Regression Equation and Correlation for Optimal Threshold Fill-factor

Keys Per Row	Regression Formula	Correlation
8	-3.1502x+0.8929	0.6735
16	-2.8352x+0.9217	0.8051
32	-2.3248x+0.9417	0.8755
64	-2.0067x+0.9599	0.8914
128	-1.8728x+0.9768	0.9466
256	-1.5105x+0.9853	0.9563
512	-1.2888x+0.9891	0.9713

For our other critical measure, **maximum throughput fill-factor**, we can see how quickly performance is lost as illustrated in Table 6.

Table 6 -- Throughput versus Fill-factor with different number of keys per page



As above, we find a linear relationship with high correlation, as seen in Figure 1. This relationship approximates the step function as illustrated in Figure 17. Looking closely at this last figure, you will see that the step values curve, we are estimating where the curve (fortunately) happen to be flat.

Figure 16 -- Fill-factor Regression against Growth Rate

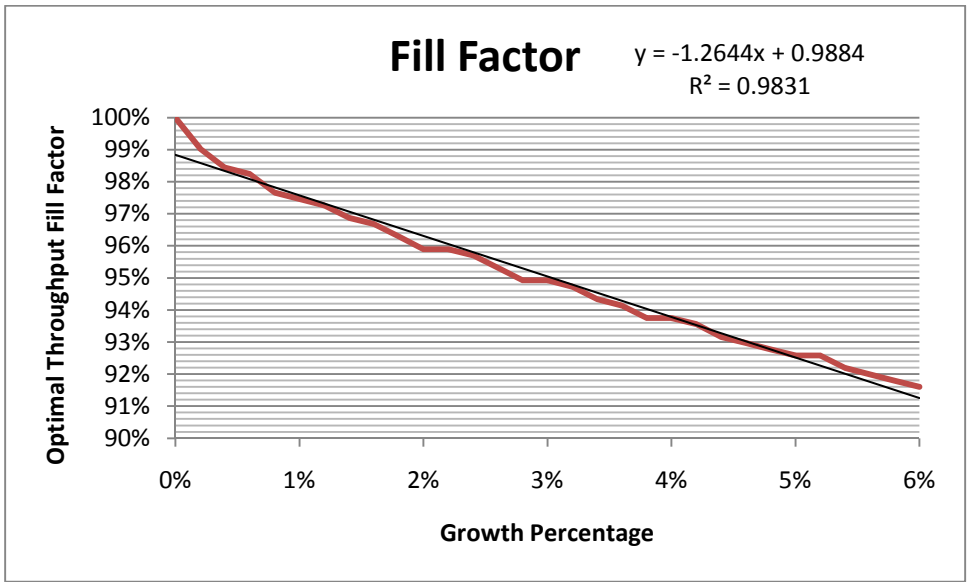


Table 7 shows the regression lines and correlations (which are stronger than those in Table 5).

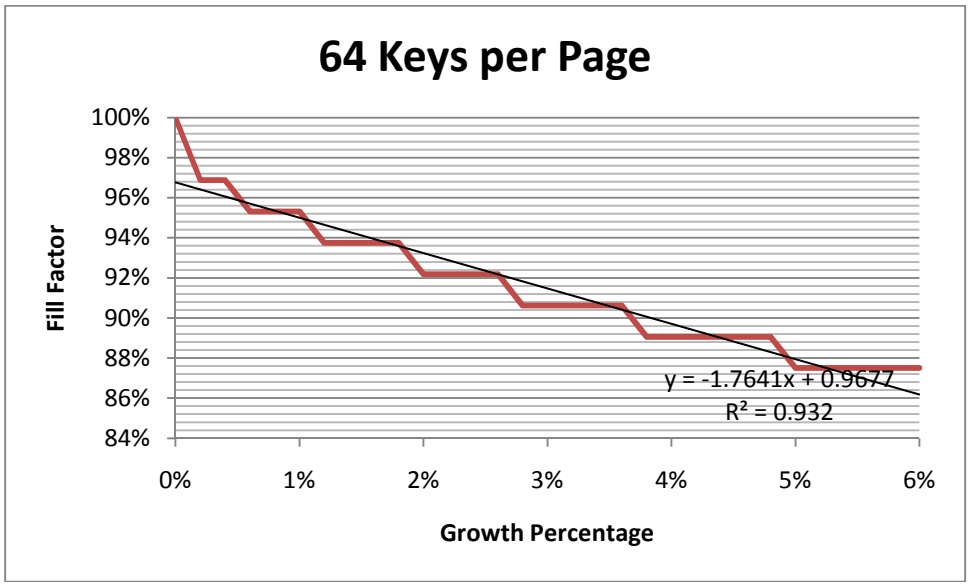
Table 7 -- Regression Equation and Correlation for Optimal Threshold Fill-factor

Keys Per Row	Regression Formula	Correlation
8	-3.6038x+0.9428	0.7655
16	-2.5706x+0.9420	0.8310
32	-2.0854x+0.9547	0.8893
64	-1.7641x+0.9677	0.9320
128	-1.5231x+0.9756	0.9595
256	-1.3625x+0.9827	0.9727
512	-1.2644x+0.9884	0.9831

From the above, two SQL functions resulted after doing some mathematics (see appendix for definition):

- Fn_OptimalThroughputFillFactorForRandom(keybytes, growthpercentage)
- Fn_OptimalThresholdFillFactorForRandom (keybytes, growthpercentage)

Figure 17 -- Example of Regression Line versus Step Function



Operational Use

Operation use means being able to forecast the expected growth of the index before the next defragmentation. For an expected growth, consider using the expected value forecasted plus two standard deviations. Underestimating growth has a significant performance cost compared to overestimating growth. For an example of how to record data to base a forecast on, see Appendix A.

Forecasting is a science itself. If your database manages sales, you may need to use time series analysis to do the forecast because of seasonality seen with orders. For purposes of *illustration*, we do a simplistic forecast in Appendix A that assumes daily defragmentation and use the last seven days of data.

Summary

This white paper examines the impact of fill-factor on performance and derives two functions that might be used operationally for optimizing the fill-factor. The analysis used a simplistic model. Even excluding many factors, the behavior is complex. The analysis illustrates the need to do frequent defragmentation of indexes and the need to adjust the fill-factor over time to insure optimal performance.

The magnitude of performance differences predicted above might not be seen in actual practices because we assumed no caching occurs and a specific hardware behavior. The functions derived provide specific recommendations for fill-factors based on key size and expected growth percentage, something the current literature lacks. Appendix A provides a sample implementation using a stored procedure that could be schedule with SQL Agent. Appendix B describes how you may manually tune fill-factors for complex key distributions.

Appendix C provides a TSQL script to rough-tune fill-factors based on key-size and an assumed 3% growth rate. Regardless of the fill-factor, the need to defragment indexes at least once a week is shown by the charts above.

Actual performance improvements due to adjusting the fill-factor may only be determined by load stressing an application (to reduce the impact of caching).

Links for Further Information

SQL Server information is found in Books Online:

- SQL Server 2008 Books Online

Bibliography

- ***SQL Server MVP Deep Dives*** (Manning, 2010), Paul Nielsen, Kalen Delaney, et al.
- ***Microsoft SQL Server 2008 Bible*** (Wiley Publishing, 2009), Paul Nielsen, Mike White, Uttam Parui.
- ***Microsoft SQL Server™ 2008 Internals: The Storage Engine*** (Microsoft Press, 2009), Kalen Delaney
- ***Professional SQL Server 2008 Internals and Troubleshooting*** (Wiley, 2010), Christian Bolton, Justin Langford, Steven Wort, et al.
- ***Professional SQL Server 2008 Administration*** (Wiley, 2009), Brian Knight, Katen Parel, Steven Wort, et al.
- ***Mastering SQL Server Profiler*** (Red Gate Books 2009), Brad McGehee

Appendix A: TSQL Implementation

The stored procedure below might be scheduled to automatically adjust fill-factors. This is an example implementation based on the assumptions described above.

SQL Functions

Fill-factors can only be set to integer values, hence an integer is returned by these functions *after rounding up if there is any decimal value.*

Caveat: The functions below are valid for @GrowthPercentage values between 0.001 (0.1%) and 0.06 (6%) .

Optimal Throughput Estimator

For most applications, optimizing the throughput is the preferred estimator.

TSQL 4 -- Function: [Fn_OptimalThroughputFillFactorForRandom]

```
CREATE FUNCTION [dbo].[Fn_OptimalThroughputFillFactorForRandom]
(
	@KeyBytes float,
	@GrowthPercentage float
)
RETURNS int
AS
BEGIN
	If @KeyBytes < 2
		SET @KeyBytes=2
	If @GrowthPercentage > 0.06
		SET @GrowthPercentage = 0.06
	If @GrowthPercentage < 0.001
		SET @GrowthPercentage = 0.001


	DECLARE @FillFactor float
	DECLARE @Rate float
	DECLARE @Offset float
	Set @Rate=- 5.2312 * Power(@keybytes,-0.244) -- R=0.95
	Set @Offset=1 - 0.2193 * Power(@keybytes, - 0.462) -- R = 0.99
	Set @FillFactor= CEILING(100 *
		(@Rate * @GrowthPercentage + @Offset))
	If @FillFactor < 50
		SET @FillFactor=50
	If @FillFactor > 99
		SET @FillFactor=99
	RETURN @FillFactor
END
```

Optimal Threshold Estimator

For some applications where a performance service level agreement may be required, the threshold estimator may be applicable.

TSQL 5 -- Function: [Fn_OptimalThresholdFillFactorForRandom]

```
Create FUNCTION [dbo].[Fn_OptimalThresholdFillFactorForRandom]
(
	@KeyBytes float,
	@GrowthPercentage float
)
RETURNS int
AS
BEGIN
	If @KeyBytes < 2
		SET @KeyBytes=2
	If @GrowthPercentage > 0.06
		SET @GrowthPercentage = 0.06
	If @GrowthPercentage < 0.001
		SET @GrowthPercentage = 0.001

	DECLARE @FillFactor float
	DECLARE @Rate float
	DECLARE @Offset float
	Set @Rate=- 5.0189 * Power(@keybytes,-0.218) -- R=0.99
	Set @Offset=1 - 0.9774 * Power(@keybytes, - 0.574) -- R= 0.99
	Set @FillFactor= CEILING(100 *
		(@Rate * @GrowthPercentage + @Offset))
	If @FillFactor < 50
		SET @FillFactor=50
	If @FillFactor > 99
		SET @FillFactor=99

	RETURN @FillFactor
END
GO
```

SQL Tables

Two tables are used. One table is used to identify the indexes that should have dynamic fill-factors. The second table tracks fill-factors, growth rate and fragmentation.

TSQL 6 -- Create Tables for Tracking Statistics

```
CREATE TABLE [dbo].[AutoFillFactor](
	[IndexKey] [varchar](50) Not NULL,
 CONSTRAINT [PK_AutoFillFactor] PRIMARY KEY CLUSTERED
(
	[IndexKey] ASC
)
```

```
CREATE TABLE [dbo].[IndexTrackingTable](
	[DatabaseName] [nvarchar](128) NOT NULL,
	[TableName] [sysname] NOT NULL,
```

```
	[SchemaName] [sysname] NOT NULL,
	[IndexName] [sysname] NOT NULL,
	[RecordedAt] [datetime] NOT NULL,
	[IndexKey] [varchar](50) NULL,
	[KeySize] [int] NOT NULL,
	[IndexBytes] [float] NOT NULL,
	[FillFactor] [int] NOT NULL,
	[GrowthPercentage] [float] NULL,
	[FragmentationPercentage] [float] NOT NULL,
	[ScanTimeEstimate]  AS ((0.5)*(100 -
	[FragmentationPercentage])+(9)*[FragmentationPercentage]
) CONSTRAINT [PK_IndexTrackingTable] PRIMARY KEY CLUSTERED
(
	[DatabaseName] ASC,
	[TableName] ASC,
	[SchemaName] ASC,
	[IndexName] ASC,
	[RecordedAt] ASC
)
```

SQL Stored Procedure

The following stored procedure records significant statistics. Then it creates dynamic TSQL using a optimal fill-factor forecast function and executes it.

TSQL 7 -- Stored Procedure: p_OptimizeFillFactor

```
CREATE proc p_OptimizeFillFactor
As
Declare @RecordedAt datetime
Declare @MyTable varchar(255)
Declare @MyIndex varchar(255)
Declare @MyFillFactor varchar(3)
Declare @MySql nvarchar(max)

Set @RecordedAt=GetDate()
INSERT INTO [IndexTrackingTable]
           ([DatabaseName]
           ,[TableName]
           ,[SchemaName]
           ,[IndexName]
           ,[RecordedAt]
           ,[IndexKey]
           ,[KeySize]
           ,[IndexBytes]
           ,[FillFactor]
           ,[FragmentationPercentage])
Select
	db_name([database_id]) as [DatabaseName],

	T.[Name] as [TableName],
	Sch.[Name] as [SchemaName],
	Indexes.[Name] as [IndexName],
	@RecordedAt,
```

```
      cast(S.object_id as varchar(11))+'_'+cast(S.index_id as
varchar(11)) as [IndexKey],
      S.[KeySize],
      S.[IndexBytes],
      Indexes.fill_factor,
      S.[FragmentationPercentage]
FROM (Select
            [database_id],
            [object_id],
            [index_id],
            max(max_record_size_in_Bytes) as [KeySize],
            sum(avg_record_size_in_bytes * record_count) as
[IndexBytes],
            Sum(avg_fragmentation_in_percent *
page_count)/Sum(page_count) as [FragmentationPercentage]
      FROM sys.dm_db_index_physical_stats(
    db_id(), NULL, NULL, NULL, 'DETAILED')
    Where index_type_desc IN ('CLUSTERED INDEX','NONCLUSTERED INDEX')
AND alloc_unit_type_desc='IN_ROW_DATA'
Group by [database_id],[object_id],[index_id]
HAVING Sum(page_count) > 0
) S
Join Sys.objects T
      On S.[object_id] = T.[object_id]
Join Sys.schemas sch
      On Sch.[schema_id] = T.[schema_id]
Join Sys.Indexes
      On Indexes.[Object_id] = S.[Object_Id]
      And Indexes.[Index_id] = S.[Index_id]

Update [IndexTrackingTable]
      Set GrowthPercentage=Case when [EstStDev]*2+[EstAvg] < 0.001 Then
0.001
            Else [EstStDev]*2+[EstAvg]  End
From [IndexTrackingTable]
JOIN (
select
      [IndexTrackingTable].[IndexKey],
      Stdev([IndexBytes])/Max([IndexBytes]) As [EstStDev],
      (Max([IndexBytes])-
Min([IndexBytes]))/Min([IndexBytes])/Count([IndexBytes]) as [EstAvg]
from [IndexTrackingTable]
JOIN [AutoFillFactor]
ON [IndexTrackingTable].[IndexKey]=[AutoFillFactor].[IndexKey]
Where RecordedAt > GetDate()-7
and [IndexBytes] is not null
and [IndexBytes] > 0
group by [IndexTrackingTable].[IndexKey] ) Estimates
ON [Estimates].[IndexKey]=[IndexTrackingTable].[IndexKey]
And [RecordedAt]=@RecordedAt

Create table #temp (mytable varchar(255), myIndex varchar(255),
MyFillFactor varchar(3))
insert into #temp (mytable,myIndex,MyFillFactor)
SELECT '['+SchemaName+'].['+TableName+']', [IndexName],
Cast([dbo].[Fn_OptimalThroughputFillFactorForRandom](KeySize,GrowthPerc
entage)as varchar(3))
```

```
FROM [IndexTrackingTable]
WHERE [RecordedAt]=@RecordedAt
      And [IndexKey] in (Select [IndexKey] From [AutoFillFactor])

DECLARE Index_cursor CURSOR FOR
SELECT mytable,myindex,myfillfactor
FROM #temp

OPEN Index_cursor
FETCH NEXT FROM Index_cursor
INTO @MyTable, @MyIndex, @MyFillFactor
WHILE @@FETCH_STATUS = 0
BEGIN
      SET @MySql='ALTER INDEX ['+@MyIndex+'] ON '+@MyTable+' REBUILD
WITH (PAD_INDEX=ON, FILLFACTOR = '+@MyFillFactor+ ')'
      exec (@MySql)
      FETCH NEXT FROM Index_cursor INTO @MyTable, @MyIndex,
@MyFillFactor
END
CLOSE Index_cursor
DEALLOCATE Index_cursor
Drop table #temp
go
```

Appendix B: Dynamic Tuning of Fill-factor

The formulas above assume *uniform-random* distribution of records across the entire index. The AutoFillFactor table facilitates tuning of the fill-factor for other types of distributions. This heuristic uses the simple formula below (using 0.5 millisecond for sequential access and 4.5 milliseconds for random access as our hardware factors).

Est Scan Time= 0.5 * (100-Fragmentation) + 9 * Fragmentation

The process is to increase or decrease the fill-factor by one-step each period and see if the Est. Scan Time reduces. The Excel formula below calculates one-step:

=CEILING(100/FLOOR(8060/***KeySize***,1),1)

Examples values are shown below,

Key Size	Fill Factor Step
900	13
162	3
81	2
80	1

This approach assumes similar growth per period between defragmentation.

Appendix C: Automatic One Time Tuning of Fill-Factor

The TSQL below assumes a maximum of 3% growth factor per week with weekly defragmentation of indexes. The TSQL defrag the indexes and assigns a fill factor appropriate for these assumptions. Subsequent defragmentation will use these fill factors.

The fill-factor uses the following table calculated from the above formulae. The actual fill factor is round down to the first integer step by SQL Server, for example 97 → 87.5% (7/8)

Figure 18 -- Optimal Fill-Factor with a 3% growth rate

Computed Fill Factor	Key Size (bytes)	SQL Data Type
71	2	
75	3	
78	4	Int
79	5	
81	6	
82	7	
83	9	DateTime
84	10	
85	13	
86	16	Unique Identifier
87	20	
88	25	
89	33	
90	44	
91	61	VarChar(50)
92	89	
93	135	NVarChar(50)
94	222	
95	403	
96	848	
97	900	

TSQL 8 -- One Time Execution to Approximate Fill-Factor

```
Declare @MyTable varchar(255)
Declare @MyIndex varchar(255)
Declare @MyFillFactor varchar(3)
Declare @MyKeySize int
Declare @MySql nvarchar(max)
DECLARE Index_cursor CURSOR FOR
      select '['+ s.name+'].['+     t.name+']' as TableName,
            '['+i.name+']' as IndexName,
            sum(c.Max_length) as [KeySize]
```

```
       from sys.tables t
      inner join sys.schemas s on t.schema_id = s.schema_id
      inner join sys.indexes i on i.object_id = t.object_id
      inner join sys.index_columns ic on ic.object_id = t.object_id
      inner join sys.columns c on c.object_id = t.object_id
            and ic.column_id = c.column_id
      where
      -- If first column is an identity we skip it
      i.object_id NOT IN (
            Select i.object_id from sys.indexes i
            inner join sys.index_columns ic
            on ic.object_id = i.object_id
            inner join sys.columns c on c.object_id = i.object_id
            and ic.column_id = c.column_id
            Where c.Is_Identity=1 and ic.index_column_id=1)
       -- clustered & nonclustered only
      and i.type in (1, 2)
      -- must be a key column
      and ic.key_ordinal > 0
      group by s.name,t.name,i.name

OPEN Index_cursor
FETCH NEXT FROM Index_cursor
INTO @MyTable, @MyIndex, @MyKeySize
WHILE @@FETCH_STATUS = 0
BEGIN
If @MyKeySize <=900
      Set @MyFillFactor='97'
If @MyKeySize <=848
      Set @MyFillFactor='96'
If @MyKeySize <=403
      Set @MyFillFactor='95'
If @MyKeySize <=222
      Set @MyFillFactor='94'
If @MyKeySize <=135
      Set @MyFillFactor='93'
If @MyKeySize <=89
      Set @MyFillFactor='92'
If @MyKeySize <=61
      Set @MyFillFactor='91'
If @MyKeySize <= 44
      Set @MyFillFactor='90'
If @MyKeySize <= 33
      Set @MyFillFactor='89'
If @MyKeySize <= 25
      Set @MyFillFactor='88'
If @MyKeySize <=20
      Set @MyFillFactor='87'
If @MyKeySize <=16
      Set @MyFillFactor='86'
If @MyKeySize <=13
      Set @MyFillFactor='85'
If @MyKeySize <=10
      Set @MyFillFactor='84'
If @MyKeySize <= 9
      Set @MyFillFactor='83'
If @MyKeySize <= 7
```

```
      Set @MyFillFactor='82'
If @MyKeySize <=6
      Set @MyFillFactor='81'
If @MyKeySize <=5
      Set @MyFillFactor='79'
If @MyKeySize <= 4
      Set @MyFillFactor='78'
If @MyKeySize <= 3
      Set @MyFillFactor='75'
If @MyKeySize <= 2
      Set @MyFillFactor='71'
      SET @MySql='ALTER INDEX '+@MyIndex+' ON '+@MyTable+' REBUILD WITH
(PAD_INDEX=ON, FILLFACTOR = '+@MyFillFactor+ ')'
      exec (@MySql)
      FETCH NEXT FROM Index_cursor INTO @MyTable, @MyIndex, @MyKeySize
END
CLOSE Index_cursor
DEALLOCATE Index_cursor
```