SQL ServerCentral.com
SQL Provision NEW Provision databases efficiently and keep data safe TRY FREE
a community service from redgate
Try SQL Clone

Daily SQL Articles by email:
your@email.com
Sign up

Write For Us
Find out more

Proactively monitor your growing SQL Server environment.
Try free
SQL Monitor

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: G+ 0
Rate this | Join the discussion | Add to briefcase

Index Fragmentation in SQL Server: Stairway to SQL Server Indexes Level 11

By David Durant, 2013/11/27 (first published: 2012/02/01)

The Series

This article is part of the Stairway Series: Stairway to SQL Server Indexes

Indexes are fundamental to database design, and tell the developer using the database a great deal about the intentions of the designer. Unfortunately indexes are too often added as an afterthought when performance issues appear. Here at last is a simple series of articles that should bring any database professional rapidly 'up to speed' with them

In Level 10 we looked at the internal structure of an index. Given this knowledge of index structure, we can examine index fragmentation: causes, cures, and when not to care.

Some Background / Review Information

The following knowledge is essential to understanding index fragmentation. Some of this information has been presented in earlier levels, in the context of using indexes to retrieve data. In this level, the context is index maintenance; thus, some old information is repeated here, and some new information is added.

Every table is either a heap or a clustered index.

If the table is a heap, then the bookmark portion of the nonclustered indexes is a row identifier (RID). If the table is a clustered index, then the bookmark portion of the nonclustered indexes is the clustered index's key values.

Index entries are stored in pages. Eight physically consecutive pages comprise an extent.

Index levels are numbered from the bottom up, starting at zero. Thus, the leaf level is always Level 0 and the lowest intermediate level is always Level 1. The root page will be a level unto itself, and will have the highest level number.

To scan an entire index means each page at the leaf level must be read once and only once. Each page contains a pointer to the next page and to the previous page. The logical sequence and

Related Articles
FORUM index fragmentation
2013/05/13
index fragmentation
FORUM Clustered Index Fragmentation
2010/02/24
Clustered Index Fragmentation
FORUM Index Vs Select criteria sequence
BLOG
FORUM
Tags
indexing
stairway series
Write For Us
Find out more

Fresh articles daily:
Get the SQL Server Central newsletter and get a new SQL Server article each day. Get Database Weekly for a roundup of all the biggest SQL news from around the web.
your@email.com
Sign up
No thanks

Copyright © 2002-2018 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

the physical sequence of pages may differ. The pointer to the next page might point to a page in the same extent, or to a page that resides hundreds of pages in front of, or behind, the current page, or even to a page in a different file.

The greater the correlation between the physical and logical sequence of pages, the fewer the number of IOs required to scan the index. When the two sequences are aligned, SQL Server can read an entire extent, or more, per IO. Also, the greater the correlation between the physical and logical sequence of pages, the more *read-ahead* reads can be used, bringing pages into memory before they are needed. This is true whether the entire index, or just a section of the index, is being scanned.

To retrieve a single index entry, one page from each level, starting with the root page, must be accessed. The performance of this operation is not affected by the correlation between an index's logical and physical sequences.

With this information in mind, we can examine the subject of index *fragmentation*.

Index fragmentation comes in two varieties – *internal fragmentation* and *external fragmentation*. The best tool for determining the fragmentation, either internal or external, of an index is the *sys.dm_db_index_physical_stats* dynamic management function. Because this function displays the index id, but not the index name, queries often join it with the *sys.indexes* view, so as to include the index name in the output. We'll be using this function throughout this Level, as first we cover internal fragmentation and then address external fragmentation.

Every page can hold a certain number of entries. That does not mean that a page always holds the maximum number of entries. Usually an index page is not completely full, for reasons that will be covered in later in this Level. When we say that an index has *internal fragmentation*, we mean that the pages are not completely full. The average amount of occupied space per page in an index is a measure of the *internal fragmentation* of the index, also referred to as the *average page fullness* of the index. Note that the higher the measurement, the lower the internal fragmentation; a page that is 100% full has no internal fragmentation.

Internal fragmentation is normally reported as a percentage, and it indicates the fullness in bytes, not in entries. Thus, an index page that has an internal fragmentation of 96% might be completely full. That is, 4% of the page might not be enough empty space for a new entry to be added. When the space occupied by page header information and page offset pointers is accounted for, a page whose individual index entries are relatively large might be "full" at 90%, 80% or even less...

The *sys.dm_db_index_physical_stats* function reports internal fragmentation in its *avg_page_space_used_in_percent* output column. The query shown in Listing 1, which examines the *SalesOrderDetail* table's clustered index, displays internal fragmentation information:

```
SELECT IX.name AS 'Name'
     , PS.index_level AS 'Level'
     , PS.page_count AS 'Pages'
     , PS.avg_page_space_used_in_percent AS
  FROM sys.dm_db_index_physical_stats(
          DB_ID(),
          OBJECT_ID('Sales.SalesOrderDetail
          DEFAULT, DEFAULT, 'DETAILED') PS
  JOIN sys.indexes IX
    ON IX.OBJECT_ID = PS.OBJECT_ID AND IX.in
  WHERE IX.name = 'PK_SalesOrderDetail_Sales
GO
```

Listing 1: A query to return internal fragmentation values

The query returns one row of output for each level in the index, as shown in Figure 1:

Name	Level	Pages	Page Fullness (%)
PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	0	1234	99.64782554979
PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	1	4	64.770200148258
PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	2	1	0.815418828762046

Zoom in | Open in new window

Figure 1: Output from the internal fragmentation query

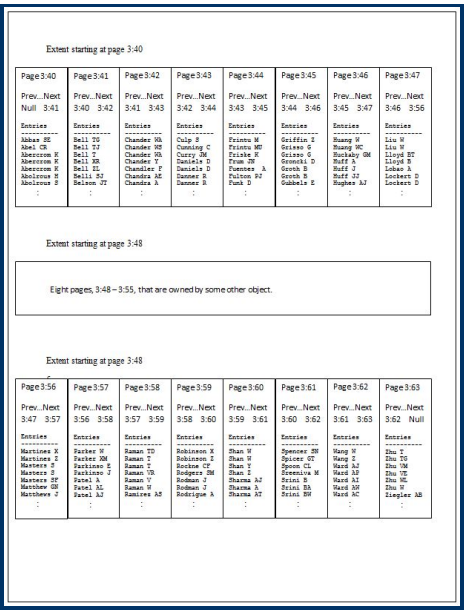
The subject of this particular query is a clustered index; thus, the leaf level of the index is the rows of the table. The output row for level 0, the first row in the above output, tells us that the table's rows are spread over 1234 pages which are, on the average, 99% full. In other words, this table has minimal internal fragmentation.

In contrast to internal fragmentation, *external fragmentation* refers to the lack of correlation between the logical sequence of an index and its physical sequence. It also is reported as a percentage. To quote Microsoft Technet, it is "the percentage of out-of-order pages in the leaf pages of an index. An out-of-order page is a page for which the next physical page allocated to the index is not the page pointed to by the next-page pointer in the current leaf page." Although the Technet definition limits itself to the leaf level only, you will note that *sys.dm_db_index_physical_stats* can return fragmentation information for all levels of an index.

Note that, unlike internal fragmentation, a higher number means a greater amount of external fragmentation; thus an index with an external fragmentation of 100% is completely fragmented externally. That is, there is no correlation at all between its logical and physical sequence of pages.

Consider the simple example illustrated in Figure 2. A sixteen page index occupies the extents starting at pages 40 and 56 of a database file. Except for the last page in each extent, every page's next-page pointer points to the next page in the extent. The last page in the extent at page 40 points to the first page in the extent 56. The last page in the extent at page 56 does not point to any page, for it is the last page in the index.

In this example, the number of out-of-order pages is zero. Although the eighth and ninth pages are not physically contiguous within the file, they are contiguous within the space that has been allocated to the index, and that is what counts. Thus, the external fragmentation for this example is zero.



Extent starting at page 3:40

Page 3:40	Page 3:41	Page 3:42	Page 3:43	Page 3:44	Page 3:45	Page 3:46	Page 3:47
Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next
Null 3:41	3:40 3:42	3:41 3:43	3:42 3:44	3:43 3:45	3:44 3:46	3:45 3:47	3:46 3:56
Entries	Entries	Entries	Entries	Entries	Entries	Entries	Entries
---	---	---	---	---	---	---	---
Abbas HE	Bell TS	Chander WA	Culp S	Frintu M	Griffin I	Huang W	Liu W
Abel CR	Bell TJ	Chander WH	Cumming C	Frintu MT	Grisso G	Huang WC	Liu W
Abercrom K	Bell T	Chander WA	Curry JM	Frinke K	Grisso G	Huckaby GM	Lloyd BT
Abercrom K	Bell KR	Chander Y	Daniels D	Frum JN	Gronski D	Huff A	Lloyd R
Abercrom K	Bell IL	Chandler F	Daniels D	Fuentes A	Groth B	Huff J	Lobao A
Abolrous S	Bell SJ	Chandra AK	Danner R	Fulton PJ	Groth B	Huff JJ	Lockert D
Abolrous S	Belson JT	Chandra A	Danner R	Funk D	Gubbels E	Hughes AJ	Lockert D
:	:	:	:	:	:	:	:

Extent starting at page 3:48

Eight pages, 3:48 – 3:55, that are owned by some other object.

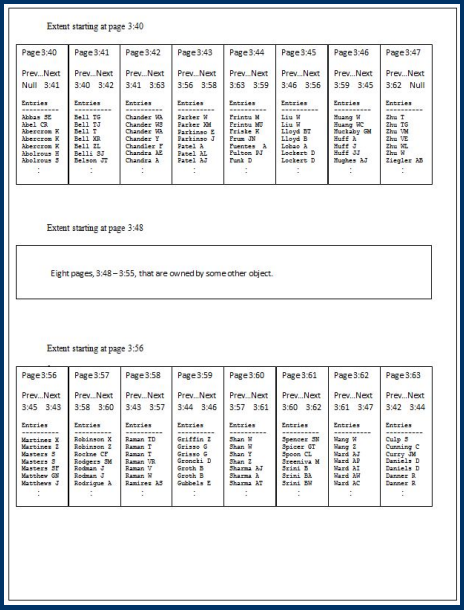
Extent starting at page 3:48

Page 3:56	Page 3:57	Page 3:58	Page 3:59	Page 3:60	Page 3:61	Page 3:62	Page 3:63
Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next
3:47 3:57	3:56 3:58	3:57 3:59	3:58 3:60	3:59 3:61	3:60 3:62	3:61 3:63	3:62 Null
Entries	Entries	Entries	Entries	Entries	Entries	Entries	Entries
---	---	---	---	---	---	---	---
Martinez K	Parker W	Raman TD	Robinson X	Shan H	Spencer ZN	Wang W	Zhu T
Martinez Z	Parker XM	Raman T	Robinson Z	Shan H	Spicer GT	Wang Z	Zhu TG
Masters S	Parkinso E	Raman T	Rochne CF	Shan Y	Spoon CL	Ward AJ	Zhu VM
Masters S	Parkinso J	Raman VR	Rodgers ZM	Shan Z	Szemiva M	Ward AJ	Zhu VE
Masters SF	Patel A	Raman V	Rodman J	Sharma AJ	Srini B	Ward AI	Zhu ML
Matthew GH	Patel AL	Raman W	Rodman J	Sharma A	Srini BA	Ward AH	Zhu W
Matthews J	Patel AJ	Ramirez AS	Rodrigue A	Sharma AT	Srini BH	Ward AC	Ziegler AB
:	:	:	:	:	:	:	:

Zoom in | Open in new window

Figure 2 –Index with no external fragmentation

On the other hand, the Figure 3 shows the same table, but with some out-of-order pages. The first three pages are logically contiguous, but after that there is no correlation at all between logical sequence and physical sequence.



Extent starting at page 3:40

Page 3:40	Page 3:41	Page 3:42	Page 3:43	Page 3:44	Page 3:45	Page 3:46	Page 3:47
Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next
Null 3:41	3:40 3:42	3:41 3:63	3:56 3:58	3:63 3:59	3:46 3:56	3:59 3:45	3:62 Null
Entries	Entries	Entries	Entries	Entries	Entries	Entries	Entries
---	---	---	---	---	---	---	---
Abbas HE	Bell TS	Chander WA	Parker W	Frintu M	Liu W	Huang W	Zhu T
Abel CR	Bell TJ	Chander WH	Parker XM	Frintu MT	Liu W	Huang WC	Zhu TG
Abercrom K	Bell T	Chander WA	Parkinso E	Frinke K	Lloyd BT	Huckaby GM	Zhu VM
Abercrom K	Bell KR	Chander Y	Parkinso J	Frum JN	Lloyd B	Huff A	Zhu VE
Abercrom K	Bell IL	Chandler F	Patel A	Fuentes A	Lobao A	Huff J	Zhu ML
Abolrous S	Bell SJ	Chandra AK	Patel AL	Fulton PJ	Lockert D	Huff JJ	Zhu W
Abolrous S	Belson JT	Chandra A	Patel AJ	Funk D	Lockert D	Hughes AJ	Ziegler AB
:	:	:	:	:	:	:	:

Extent starting at page 3:48

Eight pages, 3:48 – 3:55, that are owned by some other object.

Extent starting at page 3:56

Page 3:56	Page 3:57	Page 3:58	Page 3:59	Page 3:60	Page 3:61	Page 3:62	Page 3:63
Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next	Prev...Next
3:45 3:43	3:58 3:60	3:43 3:57	3:44 3:46	3:57 3:61	3:60 3:62	3:61 3:47	3:42 3:44
Entries	Entries	Entries	Entries	Entries	Entries	Entries	Entries
---	---	---	---	---	---	---	---
Martinez K	Robinson X	Raman TD	Griffin I	Shan H	Spencer ZN	Wang W	Culp S
Martinez Z	Robinson Z	Raman T	Grisso G	Shan H	Spicer GT	Wang Z	Cumming C
Masters S	Rochne CF	Raman T	Grisso G	Shan Y	Spoon CL	Ward AJ	Curry JM
Masters S	Rodgers ZM	Raman VR	Gronski D	Shan Z	Szemiva M	Ward AJ	Daniels D
Masters SF	Rodman J	Raman V	Groth B	Sharma AJ	Srini B	Ward AI	Daniels D
Matthew GH	Rodman J	Raman W	Groth B	Sharma A	Srini BA	Ward AH	Danner R
Matthews J	Rodrigue A	Ramirez AS	Gubbels E	Sharma AT	Srini BH	Ward AC	Danner R
:	:	:	:	:	:	:	:

Zoom in | Open in new window

Figure 3 –Index with external fragmentation

Any next-page pointer that does not point to the next physically allocated page of the index is an out-of-order pointer, regardless of whether it points forwards, backwards, within the extent, across extents, or even across files.

To illustrate how to obtain external fragmentation information for an index, we make two changes to our previous query:

Because external fragmentation concerns the relationship between pages, but not the content of the pages, it can be determined by scanning Level 1 of the index, rather than having to scan the much larger leaf level of the index. We inform SQL Server that a leaf level scan will not be necessary by changing the function's last parameter from DETAILED to LIMITED (or DEFAULT).

Also, we modify the SELECT clause to include a slightly different set of columns, those that provide external fragmentation information.

The resulting query is shown in Listing 2:

```
SELECT IX.name AS 'Name'
, PS.index_level AS 'Level'
, PS.page_count AS 'Pages'
, PS.avg_fragmentation_in_percent AS 'Exter
, PS.fragment_count AS 'Fragments'
, PS.avg_fragment_size_in_pages AS 'Avg Fra
FROM sys.dm_db_index_physical_stats(
DB_ID(),
OBJECT_ID('Sales.SalesOrderDetail'),
DEFAULT, DEFAULT, 'LIMITED') PS
JOIN sys.indexes IX
ON IX.OBJECT_ID = PS.OBJECT_ID AND IX.index_
WHERE IX.name = 'PK_SalesOrderDetail_SalesO
GO
```

Listing 2: A query to return external fragmentation

Because we specified LIMITED for the last parameter, we will get only one row of output, shown in Figure 4:

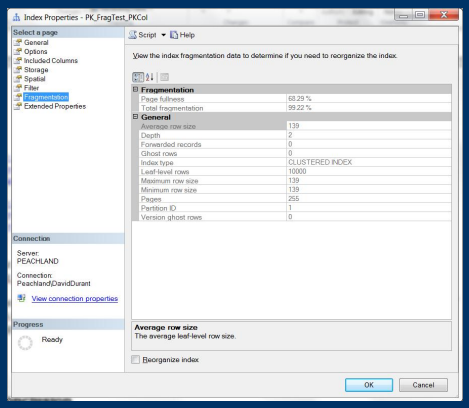
Name	Level	Pages	External Fragmentation (%)	Fragments	Avg Fragment Size
PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	0	1234	0.0810372771474878	20	61.7

Zoom in | Open in new window

Figure 4: Output to the external fragmentation query

The results tell us that our index is 1234 pages in size, consisting of 20 fragments with an average size of 61.7 pages each. Just as one cut through a piece of wood results in two pieces, and two cuts results three pieces; one out-of-sequence next-page pointer results in two fragments, two out-of-sequence next-page pointers results in three fragments, and so on. Therefore, our index has 20-1 = 19 pages that have out-of-sequence next-page pointers. On average, when scanning this index in logical sequence, 61contiguous pages will occur between each out-of-sequence page. This is an index with very little external fragmentation.

Management studio's Index Properties window, shown in Figure 5, obtains its information by doing a query similar to the one we have been doing. It refers to internal and external fragmentation by the terms *Page fullness* and *Total fragmentation* respectively.



Zoom in | Open in new window

Figure 5: SQL Server Management Studio's Index Properties window

To discuss the cause of fragmentation, and the techniques for avoiding and removing fragmentation, we must steal some information from two upcoming topics of this Stairway: information about creating / altering an index, and about inserting rows into an indexed table.

As we cover this information, remember what was mentioned in the background section of this Level: fragmentation is undesirable only if your application is scanning an index, be it a complete or partial scan. Fragmentation is not a problem when your application is retrieving a small number of entries for a single index key value, as in the five row query shown in Listing 3:

```
SELECT *
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 56080;
GO
```

Listing 3: A highly selective query

When you create or rebuild an index on a table that has been populated, a reallocation of disk space occurs, resulting in an index that has little or no external fragmentation. The amount of internal fragmentation is determined the value you specify for the *FILLFACTOR* option, which determines page fullness. No attempt is made by SQL Server to preserve this page fullness during subsequent data modifications. As rows are inserted into a table, and the resulting entries are inserted into the index pages, those pages will become ever fuller.

Eventually SQL Server may attempt to insert an entry into a page that is already full. When this happens, SQL Server will search its allocation structures to find an empty page. Once it has found an empty page, it will do one of three things, each dependent on the sequence in which new index keys are being inserted:

Random Sequence: Normally, SQL Server will move half the entries (those of higher key value) from the full page to the empty page, and then insert the new entry into the appropriate page, thus producing two pages that are half full. This technique is called *page splitting*, and is the reason why indexes of this type are known as *balanced tree* indexes. Since there is little chance that the new page is physically adjacent to the old page, there will be an increase in external fragmentation. If your application continues to insert, but not delete, rows, these two pages will eventually grow from half full to full, where upon they will split into pages that are half full. Over time, every page cycles from half full to full, again and again, resulting in an internal fragmentation (average page fullness) of approximately 75%.

Ascending Sequence: If, however, SQL Server notes that the new entry would be the last entry on the full page, it will assume that rows are being inserted into the table in the same sequence as the index. In this scenario, if the index was created on the *LastName* column, and a "Kimmel" entry is to be placed at the end of the page, SQL Server assumes that the "Kimmel" row will be followed by another "Kimmel" row, or perhaps a "Kinder" row. Therefore, it places the new row, and only the new row, in the empty page. If the table was empty when the inserts began, the new page is the last page in the index, both logically and physically. Thus, the external fragmentation remains near zero. If SQL Server was correct in its assumption that rows are arriving

in index key sequence, a full page will never split. Once a page becomes full, it stays full; resulting in little or no internal fragmentation.

Descending Sequence: Conversely, if SQL Server notes that the new entry would be the first entry on the full page, it will assume that rows are being inserted into the table in descending sequence. In this situation, SQL Server assumes that a "Kimmel" row will be followed by another "Kimmel" row or perhaps a "Kimato" row. Once again, it places the new row, and only the new row, in the empty page. The increase in external fragmentation is the same as for the random sequence scenario. But the internal fragmentation is the same as the ascending sequence scenario, at or near 100%.

Once new the entry is placed into a page, some cleanup must be done. Four next-page / previous-page pointers, spread over three pages, must be updated; and an entry pointing to the new page must be inserted at the next higher level of the index, which, in turn, might result in page split at that index level.

So, although page splitting is beneficial because it keeps the index balanced, unnecessary page splitting should be avoided; this topic will be covered later in this Level.

We can verify the three behaviors mentioned above with three simple demos. Each one will use a simple two column table that has an index on one column that you can create by running the code in listing 4.

```
USE AdventureWorks;
GO
CREATE TABLE dbo.FragTest
( PKCol    INT NOT NULL
, InfoCol  NCHAR(64) NOT NULL
, CONSTRAINT PK_FragTest_PKCol PRIMARY KEY N
);
GO
```

Listing 4: A simple table with 1 index

In each demo we insert 50,000 rows into the table in one of the three possible sequences, random, ascending and descending. At the end of each load, we view the fragmentation using the query shown in Listing 5.

```
SELECT IX.name AS 'Name'
, PS.index_level AS 'Level'
, PS.page_count AS 'Pages'
, PS.avg_page_space_used_in_percent AS 'Page
, PS.avg_fragmentation_in_percent AS 'Extern
, PS.fragment_count AS 'Fragments'
, PS.avg_fragment_size_in_pages AS 'Avg Frag
FROM sys.dm_db_index_physical_stats( DB_ID(
, DEFAULT, DEFAULT
, 'DETAILED') PS
JOIN sys.indexes IX
ON IX.OBJECT_ID = PS.OBJECT_ID AND IX.index_
WHERE IX.name = 'PK_FragTest_PKCol';
GO
```

Listing 5: The fragmentation query