SQL ServerCentral.com
SQL Prompt Code analysis as you type
SQLQuery1.sql
Try it now
redgate
a community service from redgate
Try SQL Clone

Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

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

Improve your SQL Server knowledge daily with more articles by email.
your@email.com
Sign up

Index Internal Structure in SQL Server: Stairway to SQL Server Indexes Level 10

By David Durant, 2012/01/20

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 previous levels we have taken a logical approach to indexes, focusing on what they can do for us. Now it is time to take a physical approach and examine the internal structure of indexes; for an understanding of index internals leads to an understanding of index overhead. Only by knowing index structure, and how it is maintained, can you understand, and minimize, the cost of index creation, alteration, and removal; and of row insertion, update, and deletion.

Therefore, beginning with this level, we expand our focus to include the costs of indexes, as well as the benefits of indexes. After all, minimizing the costs is part of maximizing the benefits; and maximizing the benefits of your indexes is what this Stairway is all about.

Leaf and Non-leaf Levels

The structure of any index consists of the *leaf level*and the *non-leaf levels*. Although we never explicitly said so, all previous Levels focused on the leaf level of an index. Thus, it is the leaf level of a clustered index that is the table itself; each leaf level entry being a row of the table. For a nonclustered index, it is the leaf level that contains one entry per row (except for filtered indexes), each entry consisting of the index key columns, optional included columns, and the bookmark, which is either the clustered index key columns or a RID (Row ID) value.

An index entry is also called an index row; regardless of whether it is a table row (clustered index leaf level entry), refers to a table

Related Articles

FORUM index entry
2008/10/24
index entry

ARTICLE Clustered Columnstore Index Gives "Unable to find index entry" Error
2014/08/28
The Clustered columnstore index generates "unable to find index entry" error and a memory dump after...

FORUM Isolation level and Index
2
D

ARTICLE In
S
L
2
In
in
v

ARTICLE E
In
In
2
A nonclustered index entry consists of search key columns, included columns, and the bookmark. The b...

Stay up to date:
Daily newsletters with brand new articles, scripts, editorials and a Question of the Day help you keep on top of SQL Server.
your@email.com
Sign up
No thanks

Tags
indexing
stairway series

Write For Us
Find out more

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

row (nonclustered index leaf level), or points to a page at a lower level (non-leaf levels).

The non-leaf levels are a structure built on top of the leaf level, which enable SQL Server to:

- Maintain the index's entries in index key sequence.
- Quickly find a leaf level row given the index key value.

In Level 1, we used the phone book as an analogy to help explain the benefits of an index. Our phone book user, who was looking for "Meyer, Helen", knew that entry would be near the middle of any sorted list of last names, and jumped directly to the middle of the white pages to begin the search. SQL Server, however, has no such intrinsic knowledge of English language last names, or of any other data. Nor would it know which page was the "middle" page, unless it walked the entire index from beginning to end. So SQL Server builds some additional structure into the index.

This additional structure is called the non-leaf levels, or *node levels*, of the index; and is thought of as being built on top of the leaf level, regardless of where its pages are physically located. Its purpose is to give SQL Server a single page entry point for every index, and a short traversal from that page to the page containing any given search key value.

Each page in an index, regardless of its level, contains index rows, or entries. In the leaf level pages, as we have repeatedly seen, each entry either points to a table row or is the table row. So if the table contained one billion rows, the index's leaf level would contain one billion entries.

In the level just above the leaf level, that is, the lowest non-leaf level, each entry points to one leaf level page. If our one billion entry index averaged 100 entries per page, which is a realistic number for an index whose search key consists of a few numeric, date, and code columns, then the leaf level would contain 1,000,000,000 / 100 = 10,000,000 pages. In turn, the lowest non-leaf level would contain 10,000,000 entries, each pointing to a leaf level page, and would span 100,000 pages.

Each higher non-leaf level has pages whose entries each point to a page at the next lower level. Thus, our next higher non-leaf level would contain 100,000 entries and be 1,000 pages in size. The level above that would contain 1,000 entries and be 10 pages in size; the one above that would consist of ten entries residing on just one page, and that is where it stops.

The lone page sitting at the top of the index is called the *root page*. The levels of an index that lie below the root page level and above the leaf level are referred to as the *intermediate levels*. The numbering of the levels starts at zero and works upward from the leaf level. Thus, the lowest intermediate level is always level 1.

Non-leaf level entries contain only index key columns and the pointer to a lower level page. Included columns only exist in the leaf level entries; they are not carried in the non-leaf level entries.

Each page in an index, except the root page, contains two additional pointers. These pointers point to the next page and the previous page, in index sequence, at the same level. The resulting bi-directional chain of pages enables SQL Server to scan the pages of any level in either ascending or descending sequence.

The simple diagram, shown in Figure 1 below, helps illustrate this tree like structure of an index. This diagram represents an index created on the *LastName* / *FirstName* columns of a theoretical *Personnel.Employee* table, using the following SQL:

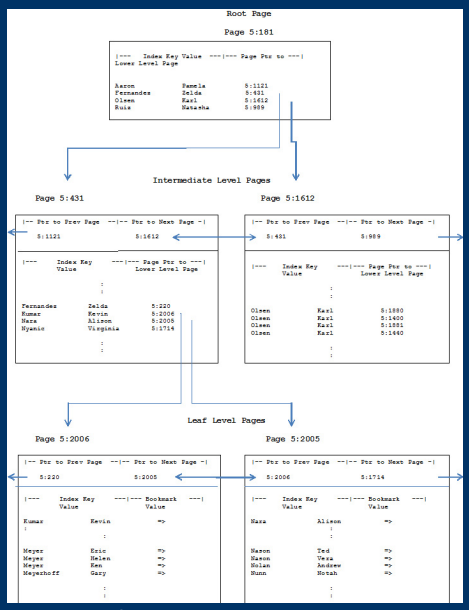
```
CREATE NONCLUSTERED INDEX IX_Full_Name
ON Personnel.Employee
(
LastName,
FirstName,
)
GO
```

Diagram Notes:

A pointer to a page consists of the database file number plus the page number. Thus a pointer value of *5:4567* points to the 4567th page of database file #5.

Most of the sample values have been taken from the *Person.Contact* table in the *AdventureWorks*database. A few others have been added for illustrative purposes.

Karl Olsen is the most popular name in the sample. There are so many Karl Olsens that their entries span an entire intermediate level index page.



🔍 Zoom in | Open in new window

Figure 1 – A Vertical Slice of an Index

For the sake of clarity and illustration, the diagram differs from a typical index in the following ways:

The number of entries per page in a typical index would be greater than the number shown in the diagram, and, thus, the number of pages for every level except the root would be greater than is shown. The leaf level, especially, would have far more pages than can be shown in our space limited diagram.

The entries of an actual index are not sequenced on the page. It is the page's entry offset pointers that provide sequenced access

to the entries. (See *Level 4 – Pages and Extents* for more information regarding offset pointers.)

There can be, and often is, more correlation between the physical and logical sequence of an index than is shown in the diagram. This lack of correlation between the physical and logical sequence of an index is called *external fragmentation*, and is discussed in *Level 11 – Fragmentation*.

As mentioned earlier, an index can have more than one intermediate level.

It is as if our white pages user, looking for Helen Meyer, opened the phone book and discovered that the first page, and only the first page, was pink. Within the pink page's list of sequenced entries was one that said "For names between "Fernandez, Zelda" and "Olsen, Karl" see blue page S.431". When our user went to blue page S.431, one of the entries on that page said "For names between "Kumar, Kevin" and "Nara, Alison" see white page S.2806". The pink page would correspond to the root, the blue pages to the intermediate level, and the white pages are the leaf.

The location of the root page is stored in a system table along with other information about the index. Whenever SQL Server needs to access the index entry that matches an index key value, it starts at the root page and works its way through one page at each level in the index until it reaches the leaf level page that contains the entry for that index key. In our one billion row table example, five page reads would take SQL Server from the root page to the leaf level page and its desired entry; in our diagramed example, three reads would suffice. In a clustered index, this leaf level entry would be the actual data row; in a nonclustered index, this entry would contain either the clustered index key columns or a RID value.

The number of levels, or *depth*, of an index will depend upon the size of the index key and the number of entries. In the *AdventureWorks* database, no index has a depth greater than three. In databases with very large tables or very wide index key columns, depths of six or greater may occur.

The *sys.dm_db_index_physical_stats* function provides information on your indexes, including index type, depth, and size. It is a *table-valued* function which can be queried. The example shown in Listing 1 returns summary information for all indexes of the SalesOrderDetailtable.

```
SELECT OBJECT_NAME(P.OBJECT_ID) AS 'Table'
     , I.name AS 'Index'
     , P.index_id AS 'IndexID'
     , P.index_type_desc
     , P.index_depth
     , P.page_count
  FROM sys.dm_db_index_physical_stats (DB_I
                                       OBJE
                                       NULL
  JOIN sys.indexes I ON I.OBJECT_ID = P.OBJ
                    AND I.index_id = P.inde
```

Listing 1: Querying the sys.dm_db_index_physical_stats functionThe results shown in Figure 2:

Table	Index	IndexID	index_type_desc	index_de...	page_co...
SalesOrderDetail	PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	1	CLUSTERED INDEX	3	1234
SalesOrderDetail	AK_SalesOrderDetail_rowguid	2	NONCLUSTERED INDEX	3	407
SalesOrderDetail	IX_SalesOrderDetail_ProductID	3	NONCLUSTERED INDEX	2	226

🔍 Zoom in | Open in new window

Figure 2: The results of querying the sys.dm_db_index_physical_stats function

Conversely, the code shown in Listing 2 requests detailed information for a specific index, the nonclustered index on the table's *uniqueidentifier* column of the SalesOrderDetail table. It returns one row per index level, as shown in Figure 3.

Listing 2: Querying sys.dm_db_index_physical_stats for detailed information.

```
SELECT OBJECT_NAME(P.OBJECT_ID) AS 'Table'
     , I.name AS 'Index'
     , P.index_id AS 'IndexID'
     , P.index_type_desc
     , P.index_level
     , P.page_count
  FROM sys.dm_db_index_physical_stats (DB_I
  JOIN sys.indexes I ON I.OBJECT_ID = P.OBJ
                    AND I.index_id = P.inde
```

Figure 3: The results of querying sys.dm_db_index_physical_stats for detailed information

Table	Index	IndexID	index_type_desc	index_level	page_count
SalesOrderDetail	AK_SalesOrderDetail_rowguid	2	NONCLUSTERED INDEX	0	407
SalesOrderDetail	AK_SalesOrderDetail_rowguid	2	NONCLUSTERED INDEX	1	2
SalesOrderDetail	AK_SalesOrderDetail_rowguid	2	NONCLUSTERED INDEX	2	1

🔍 Zoom in | Open in new window

From the results shown in Figure 3, we can see that:

- The leaf level of this index is spread over 407 pages.
- The one and only intermediate level requires just two pages.
- The root level is, as always, a single page.

The size of the non-leaf portion of an index is typically one tenth to one two-hundredth the size of the leaf level; depending upon which columns comprise the search key, the size of the bookmark, and which, if any, included columns are specified. In other words, indexes are, relatively speaking, very wide and very short. This is different from most sample diagrams of indexes, such as the one in Figure 1, which tend to be tall and narrow.

Remember, included columns are only applicable to nonclustered indexes and they only appear in the leaf level entries; they are omitted from the higher level entries, which is why they do not add to the size of the non-leaf levels.

Because the leaf level of a clustered index is the data rows of the table, only the non-leaf portion of a clustered index is additional information, requiring additional storage. The data rows would exist whether the index were created or not. Thus, creating a clustered index may take time and consume resources, but when

the creation is finished, very little extra space is consumed in the database.

The structure of index enables SQL Server to quickly access any entry for a specific index key value. Once that entry has been found, SQL Server can:

- Access the row for that entry.
- Traverse the index from that point in either ascending or descending sequence.

This indexing tree structure has been in use for a long time, longer even than relational databases, and it has proven itself over time.

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

Sign up to our RSS feed and get notified as soon as we publish a new level in the Stairway!

Improve your SQL Server knowledge daily with more articles by email.

your@email.com

Sign up

Thank this author by sharing:

0

Rate this | Join the discussion | Add to briefcase

Total article views: 13506 | Views in the last 30 days: 7