SQL ServerCentral.com
redgate | Hub Insights Learning Forums Ideas Discover the new Redgate Hub
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

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

Bookmarks in SQL Server Indexes: Stairway to SQL Server Indexes Level 6

By David Durant, 2011/08/03

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 the preceding level, we have looked at indexes as a set of sequenced entries; one entry per row. We have emphasized the logical aspects of indexes more than the physical, concentrating on the data that is in each entry and on the impact of having that data in index key sequence. Thus far, we have covered the first two components of a nonclustered index entry; the **search key** and the **included columns**. In this level, we examine the third, and last, component; **the bookmark**.

What's in the Bookmark?

The bookmark has been mentioned before, but only to say that it enables SQL Server to "quickly" navigate from the nonclustered index entry to the corresponding row. Now, it is time to go into more detail regarding these bookmarks, for the content of the bookmark differs depending upon whether the underlying table is a heap or a clustered index.

Regardless of whether the table is a heap or a clustered index, each row in a table is the nth row on its page. That page is the nth page in its data file. And that file is the nth file in the set of files that comprise the database. Therefore, each row in a database, at any given point in time, can be identified by three numbers: **file number - page number - row number**. This identifying composite of three numbers is called the row id, usually shortened to RID. Most tools that display SQL Server internals information will display these three numbers separated by colons (instead of hyphens). So the 12th row on the 77th page of file 1 would have a RID of 1:77:12.

Generally speaking, the rows of a heap do not move; once they have been inserted into a page they remain on that page. To be

Related Articles
FORUM Clustered Index - sys.dm_db_index_usage_stats
2008/06/17
Bookmark Lookup on a Clustered Index
FORUM Question about included columns in nonclustered indexes: Where exactly are they stored?
2012/11/19
FORUM
FORUM
FORUM

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

Tags
indexing
stairway series

Write For Us
Find out more

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

more technically-precise: rows in a heap *seldom* move, and when they do move, they leave a forwarding address at the old location. The rows of a clustered index, however, can move: that is, they can be relocated to another page during data modification or index reorganization. More details concerning what happens as rows are modified, including information about forwarding rows, will be provided in later levels.

Since the rows of a heap do not move, the RID permanently identifies each row in the heap. Not only is this value permanent, it also physically defines the location of the row. This makes it the ideal value to be used as the bookmark value in a heap's nonclustered indexes, which is why SQL Server uses it in all nonclustered indexes created on all heaps.

So, let's say the `SalesOrderDetail` table were a heap, having its rows in no specified sequence, and we then created the index that we used in Level 5, the nonclustered index on `ProductID` / `ModifiedDate` with the same included columns, as shown in Listing 6.1.

```
CREATE NONCLUSTERED INDEX FK_ProductID_Modi
ON Sales.SalesOrderDetail(ProductID, Modifi
INCLUDE (OrderQty, UnitPrice, LineTotal)
```

Listing 6.1: A non-clustered index with included columns

In the index, the sequenced entries would look like this:

```
:- Key Columns -:      :---  Included Columns  -
--:          : Bookmark :

ProductID   ModifiedDate      OrderQty UnitPrice
LineTotal
----------- ------------      -------- ---------
--------- -------------

Page n-1:

709         01 Feb 2002       1
5.70       5.70      3:9198:41
709         01 May 2002       1
5.70       5.70      3:969:2
710         01 Jul 2001       1
5.70       5.70      3:9840:29
710         01 Jul 2001       1
5.70       5.70      3:9916:29
710         01 Sep 2001       1
5.70       5.70      4:12331:32

Page n:

710         01 Oct 2001       1
5.70       5.70      3:1911:33
710         01 Nov 2001       1
5.70       5.70      4:2604:34
710         01 Nov 2001       1
5.70       5.70      4:2889:34
710         01 Nov 2001       1
5.70       5.70      3:3522:35
```

```
710         01 Nov 2001       1
5.70       5.70      3:3623:35

710         01 Jun 2002       1
5.70       5.70      4:3917:5
712         01 Jul 2001       1
       5.19       5.19      3:9886:29
712         01 Jul 2001       1
5.19       5.19      3:10270:29
712         01 Aug 2001       1
5.19       5.19      4:10609:30
712         01 Aug 2001       1
5.19       5.19      4:10617:30

Page n+1:

712         01 Aug 2001       1
5.19       5.19      4:10689:30
712         01 Aug 2001       1
5.19       5.19      4:10885:30
712         01 Aug 2001       1
5.19       5.19      4:11002:30
712         01 Sep 2001       1
5.19       5.19      4:12318:32
712         01 Sep 2001       1
5.19       5.19      4:509:32
```

The bookmark values for the rows of our index are very efficient, pointing directly to the corresponding rows; but they are dependent upon the underlying table being a heap. And although these values are very efficient for doing row lookups, they contain no information that has any value to the user.

The alternative to this type of RID-based bookmark, is the bookmark that is used when a nonclustered index is created on a table that has a clustered index; or, to be succinct and precise, the bookmark of a clustered index's non-clustered index.

If the table is a clustered index, then the rows can be relocated within the table. Therefore, for clustered indexes, the RID does not permanently identify the row, and a different value must be used as the non-clustered index bookmark value. The value that is used is the *index key of the clustered index*.

This solves the need for a consistent bookmark value. For when a row of a clustered index is moved to a new page, it is only moved, it is not modified; the clustered index key value remains unchanged. Thus, the bookmark value can always be used to retrieve the corresponding row; it just means that the row will retrieved via the clustered index rather than by its physical location.

However, this use of the clustered index key as the nonclustered index bookmark means that the clustered index key should meet three criteria:

- **It must be unique**. Each index entry bookmark must allow SQL Server to find the one row in the table that corresponds to that entry. If you create a clustered index that is not unique, SQL Server will make the clustered index unique by generating an additional value that "breaks the tie" for duplicate keys. This extra value is generated by SQL Server to

create uniqueness is called the **uniquifier** and is transparent to any client application. You should carefully consider whether or not to allow duplicates in a clustered index, for the following reasons:

  - Generating uniquifiers is extra overhead. SQL Server must decide, at insert time, if a new row's key is a duplicate of an existing row's key; and, if so, generate a uniquifier values to add to the new row.
  - The uniquifier is a meaningless piece of information; a meaningless piece of information that is being propagated into the table's nonclustered indexes. It's usually better to propagate a meaningful piece of information into the nonclustered indexes.

- **It should be short**. That is, it should contain a small number of bytes, for it will be propagated into the all nonclustered indexes. Last name / first name / middle name / street address as the clustered index for the `Contact` table might seem like a good idea, but if there are multiple nonclustered indexes on the table, it is not a good idea. If there are n nonclustered indexes on the table, then every contact's last name / first name / middle name / street address value will be stored in n+1 locations.
- **It should be static**. That is, its values should seldom change. A change in the value of the clustered index key for a row forces that row's entry in each nonclustered index to be updated with the new key value. Thus, for a table with n non-clustered indexes, a single update of a clustered index key column turns into n+1 updates (not to mention n+1 log file inserts).

When the `AdventureWorks` design team chose a clustered index for the `SalesOrderDetail` table, they adhered to all three of the guidelines. By choosing `SalesOrderID` / `SalesOrderDetailID` as the clustered index key they achieved narrow, static and unique. By adding `SalesOrderID` as the left most column of the key, even though `SalesOrderDetailID` alone is unique, they got clustering; all the rows for a single order grouped together on the same page or two. By making `SalesOrderID` / `SalesOrderDetailID` the primary key of the table as well as the clustered index key, they eliminated the need for a separate index on `SalesOrderDetailID`. No one ever asks to see line item #7 anyway; they ask to see the all line items for order #47386 or they ask to see line item #7 of order #47386. Most important of all, as we have said several times, they forced the table to be in the sequence that best served the overall application.

Now, if we create the same nonclustered index as shown in Listing 6.1, but this time on the clustered index version of the `SalesOrderDetail` table, the sequenced entries in the index would look like this:

```
:- Key Columns -:      : ---   Included Columns
---:    :---   Bookmark   ---:

ProductID   ModifiedDate      OrderQty UnitPrice
LineTotal      OrderId     DetailId
----------- ------------      -------- ---------
---------      ----------- ----------

Page n-1:
```

```
709         01 Feb 2002       1
5.70       5.70      45329       6392
709         01 May 2002       1
5.70       5.70      46047       8601
710         01 Jul 2001       1
5.70       5.70      43670       111
710         01 Jul 2001       1
5.70       5.70      43676       152
710         01 Sep 2001       1
5.70       5.70      44075       1448
```

Page n:

```
710         01 Oct 2001       1
5.70       5.70      44303       2481
710         01 Nov 2001       1
5.70       5.70      44484       2853
710         01 Nov 2001       1
5.70       5.70      44499       3006
710         01 Nov 2001       1
5.70       5.70      44523       3346
710         01 Nov 2001       1
5.70       5.70      44527       3400
710         01 Jun 2002       1
5.70       5.70      46365       10183
712         01 Jul 2001       1
5.19       5.19      43673       136
712         01 Jul 2001       1
5.19       5.19      43694       342
712         01 Aug 2001       1
5.19       5.19      43846       524
712         01 Aug 2001       1
5.19       5.19      43847       528
```

Page n-1:

```
712         01 Aug 2001       1
         5.19       5.19      43851       567
712         01 Aug 2001       1
5.19       5.19      43863       672
712         01 Aug 2001       1
5.19       5.19      43871       735
712         01 Sep 2001       1
5.19       5.19      44074       1441
712         01 Sep 2001       1
5.19       5.19      44109       1729
```

Both versions of our nonclustered index, the 'created on the heap' version shown earlier and the 'created on the clustered index' version shown here, are identical except for their bookmark values.

Is one of these designs better than the other? Maybe, but there is not much in it, either way. The RID bookmarks allow for a faster lookup of rows in the underlying table, a lookup necessitated by the failure of the index to cover the query. The index key bookmarks result in a slower lookup of rows in the underlying table, but increase the possibility that the index will cover the query, and often contain the foreign key value needed for doing a JOIN.

The real answer to the question "Which is better?" is "Neither!" When indexing a table, the most important decision is: What index, if any, should be the clustered index for the table? Once that decision has been made (in accordance with the three guidelines mentioned earlier in this level), you needn't worry about the impact of that decision on your nonclustered indexes; they'll perform just fine.

A nonclustered index entry consists of search key columns, included columns, and the bookmark. The bookmark value will be either a RID or the clustered index's key, depending upon whether the table is a heap or a clustered index. Choosing the best clustered index for a table requires that you follow three guidelines to ensure that the index key will make a good bookmark.

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!

Keep up to date with SQL Server - new articles every day.

your@email.com

Sign up

Thank this author by sharing:

0

Rate this | Join the discussion | Add to briefcase

Total article views: 13425 | Views in the last 30 days: 23