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

SQL Prompt
Code analysis as you type
SQLQuery1.sql
Try it now
redgate

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

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

Indexes with Included Columns: Stairway to SQL Server Indexes Level 5

By David Durant, 2011/07/13

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

The preceding levels introduced clustered and nonclustered indexes, highlighting the following aspects of each:

- There is always one entry in the index for each row in the table (we noted that an exception to this rule will be covered in a later level). These entries are always in index key sequence.
- In a clustered index, the index entry is the actual row of the table.
- In a nonclustered index, the entry is separate from the data row; and consists of the index key columns and a bookmark value to map the index key columns to an actual row of the table.

The last half of the previous sentence is correct but incomplete. In this level we examine the option to include additional columns to a non-clustered index, called included columns. In Level 6, which examines Bookmark operations, we will see that SQL Server might unilaterally add some columns to your index.

Included Columns

Columns that are in a nonclustered index, but are not part of the index key, are called included columns. These columns are not part of the key, and so do not impact the sequence of entries in the index. Also, as we will see, they cause less overhead than key columns.

When creating a nonclustered index, we specify the included columns separately from the key columns, as shown in Listing 5.1.

Related Articles

ARTICLE Covering Index using Included Columns
2010/02/02
This article from Josef Richberg details the benefits of included columns for use in creating a cove...

FORUM Included Columns
2009/08/07
Why not include every table field in the included columns list for every index?

FORUM Indexes with Include
2
H
h

FORUM D
in
to
2
D
c
in

BLOG M
in
2
This article is the part of series Mastering Index, in this particular article we will be explori...

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.

```
CREATE NONCLUSTERED INDEX FK_ProductID_ Mod
       ON Sales.SalesOrderDetail (ProductID
       INCLUDE (OrderQty, UnitPrice, LineTo
```

Listing 5.1: Creating a nonclustered index with included columns

In this example, `ProductID` and `ModifiedDate` are the index key columns, and `OrderQty` `UnitPrice` and `LineTotal` are the included columns.

Had we not specified the `INCLUDE` clause in the above SQL statement, the resulting index would have looked like this:

```
ProductID   ModifiedDate   Bookmark

Page n:

707         2004/07/25        =>
707         2004/07/26        =>
707         2004/07/26        =>
707         2004/07/26        =>
707         2004/07/27        =>
707         2004/07/27        =>
707         2004/07/27        =>
707         2004/07/28        =>
707         2004/07/28        =>
707         2004/07/28        =>
707         2004/07/28        =>
707         2004/07/28        =>
707         2004/07/28        =>

Page n+1:

707         2004/07/29        =>
707         2004/07/31        =>
707         2004/07/31        =>
707         2004/07/31        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
708         2001/07/01        =>
```

However, having told SQL Server to include the `OrderQty`, `UnitPrice` and `LineTotal` columns, the index looks like this:

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

ProductID   ModifiedDate      OrderQty
   UnitPrice   LineTotal

Page n-1:

707         2004/07/29        1
34.99       34.99       =>
707         2004/07/31        1
34.99       34.99       =>
707         2004/07/31        3
```

```
34.99      104.97       =>
707         2004/07/31        1
34.99       34.99       =>
708         2001/07/01        5
20.19      100.95       =>
```

```
Page n:
```

```
708         2001/07/01        1
20.19       20.19       =>
708         2001/07/01        1
20.19       20.19       =>
708         2001/07/01        2
20.19       40.38       =>
708         2001/07/01        1
20.19       20.19       =>
708         2001/07/01        2
20.19       40.38       =>
```

```
708         2001/12/01        7
20.19      141.33       =>
708         2001/12/01        1
20.19       20.19       =>
708         2002/01/01        1
20.19       20.19       =>
708         2002/01/01        1
         20.19       20.19       =>
708         2002/01/01        1
20.19       20.19       =>
```

```
Page n+1:
```

```
708         2002/01/01        2
20.19       40.38       =>
708         2002/01/01        5
20.19      100.95       =>
708         2002/02/01        1
20.19       20.19       =>
708         2002/02/01        1
20.19       20.19       =>
708         2002/02/01        2
20.19       40.38       =>
```

Examining the contents of this index shown, it is apparent that the rows are ordered by the index key columns. The five rows for product 708 with a modified date of January 1, 2002 (highlighted in bold), for example, are contiguous in the index, as are the rows every other `ProductID`/`ModifiedDate` combination.

You might ask "Why even have included columns? Why not simply add `OrderQty`,`UnitPrice` and `LineTotal` to the index key?" There are several advantages in having these columns in the index but not in the index key, such as:

- Columns that are not part of the index key do not affect the location of the entry within the index. This, in turn, reduces the overhead of having them in the index. For instance, if the `ProductID` or `ModifiedDate value` in the row is modified, then that row's entry must be relocated within the index. But, if the `UnitPricevalue` in the row is modified, the index entry still needs to be updated, but it does not need to be moved.
- The effort required to locate an entry(s) in the index is less.
- The size of the index will be slightly smaller.
- The data distribution statistics for the index will be easier to maintain.

Most of these advantages will be more meaningful in later levels, when we look at the internal structures of indexes and at some additional information that is maintained by SQL Server for optimizing query performance.

Deciding whether an index column is part of the index key, or just an included column, is not the most important indexing decision you will ever make. That said, columns that frequently appear in the `SELECT` list but not in the `WHERE` clause of a query are best placed in the included columns portion of the index.

In Level 4, we expressed agreement with the designers of the AdventureWorksdatabase regarding their decision to make `SalesOrderID SalesOrderDetailID` the clustered index of the `SalesOrderDetail` table. Most queries against this table will request data ordered or grouped by sales order number. However, a number of queries, perhaps from the warehouse staff, will need information in product sequence. These are the queries that will benefit from the index shown in Listing 5.1.

To illustrate the potential benefit of having the included columns in that index, we will look at two queries against the SalesOrderDetailtable, each of which we will execute three times, as follows:

- Run 1: No nonclustered index
- Run 2: Using a nonclustered index that contains no included columns (only the two key columns)
- Run 3: Using the nonclustered index as defined in Listing 5.1

As we have done in previous levels, we again use *number of reads* as the primary metric, but we also use SQL Server Management Studio's "Display Actual Execution Plan" option to view the plan for each execution. This will give us an added metric: the percentage of the workload that was spent on non-read activity, such as matching up related data after they have been read into memory. This gives us a better understanding of the total cost of the query.

Our first query, shown in Listing 5.2, is one that provides activity totals by date for a specific product.

```
SELECT  ProductID ,
        ModifiedDate ,
        SUM(OrderQty) AS 'No of Items' ,
        AVG(UnitPrice) 'Avg Price' ,
        SUM(LineTotal) 'Total Value'
FROM    Sales.SalesOrderDetail
WHERE   ProductID = 888
GROUP BY ProductID ,
        ModifiedDate ;
```

Listing 5.2: The "Activity totals by product" query

Since indexes can impact the performance of a query, but not the results, executing this query against the three different indexing schemes always yields the following row set:

```
ProductID   ModifiedDate    No of Rows  Avg
Price         Total Value

----------- ------------    ----------- --------
---------------------
888         2003-07-01      16
602.346           9637.536000
888         2003-08-01      13
602.346           7830.498000
888         2003-09-01      19
602.346           11444.574000
888        2003-10-01       2
602.346           1204.692000
888         2003-11-01      17
602.346           10239.882000
888         2003-12-01      4
602.346           2409.384000
888         2004-05-01      10
  602.346           6023.460000
888         2004-06-01      2
602.346           1204.692000
```

The eight rows of output are aggregated from the thirty nine `ProductID = 888` rows in the table to give one output row for each date that had one-or-more `ProductID = 888` sales.The basic scheme for conducting our test is shown in Listing 5.3. Before you run any queries, make sure you run `SET STATISTICS IO ON`

```
IF EXISTS ( SELECT  1
            FROM    sys.indexes
            WHERE   name = 'FK_ProductID_Mo
                    AND OBJECT_ID = OBJECT_
    DROP INDEX Sales.SalesOrderDetail.FK_Pr
GO

--RUN 1: Execute Listing 5.2 here (no non-c

CREATE NONCLUSTERED INDEX FK_ProductID_Modi
ON Sales.SalesOrderDetail (ProductID, Modif

--RUN 2: Re-execute Listing 5.2 here (non-c

IF EXISTS ( SELECT  1
            FROM    sys.indexes
            WHERE   name = 'FK_ProductID_Mo
                    AND OBJECT_ID = OBJECT_
    DROP INDEX Sales.SalesOrderDetail.FK_Pr
GO

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

--RUN 3: Re-execute Listing 5.2 here (non-c
```

Listing 5.3: Testing the "Activity totals by product" query

The relative effort required to execute the query against each indexing scheme is shown in Table 5.1.

Run 1: No Nonclustered Index	Table 'SalesOrderDetail'. Scan count 1, logical reads 1238. Non read activity: 8%.
Run 2: Index – No Included Columns	Table 'SalesOrderDetail'. Scan count 1, logical reads 131. Non read activity: 0%.
Run 3: With Included Columns	Table 'SalesOrderDetail'. Scan count 1, logical reads 3. Non read activity: 1%.

Table 5.1: Results of running the first query three times with different nonclustered indexes available

As you can see from these results:

- Run 1 required a complete scan of the `SalesOrderDetail` table: every row had to be read and examined to determine if it should participate in the result or not.
- Run 2 used the nonclustered index to quickly find the bookmarks for just the 39 requested rows, but it had to retrieve each of those rows individually from the table.
- Run 3 found everything that it needed in the nonclustered index, and in the most advantageous sequence – `ModifiedDate` within `ProductID`. It jumped quickly to the first requested entry, read 39 consecutive entries, doing the aggregate calculations for each entry as it was read, and was done.

Our second query is identical to the first, except for a change in the `WHERE` clause. This time the warehouse is requesting information based on date, rather than on product. We must filter on the right-most search key column, `ModifiedDate`, rather than the left-most column, `ProductID`. The new query is shown in Listing 5.4.

```
SELECT  ModifiedDate ,
        ProductID ,
        SUM(OrderQty) 'No of Items' ,
        AVG(UnitPrice) 'Avg Price' ,
        SUM(LineTotal) 'Total Value'
FROM    Sales.SalesOrderDetail
WHERE   ModifiedDate = '2003-10-01'
GROUP BY ModifiedDate ,
        ProductID ;
```

Listing 5.4: The "Activity totals by date" query

The resulting row set, in part, is:

```
ProductID   ModifiedDate    No of Items Avg
Price             Total Value
----------- ------------    ----------- --------
------------- ----------------
```

```
                                        :
                                        :
782         2003-10-01      62
1430.9937               86291.624000
783         2003-10-01      72
         1427.9937               100061.564000
784         2003-10-01      52
1376.994                71603.688000
792         2003-10-01      12
1466.01                 17592.120000
793         2003-10-01      46
1466.01                 67436.460000
794         2003-10-01      37
1466.01                 54242.370000
795         2003-10-01      22
1466.01                 32252.220000
                                        :
                                        :
(164 row(s) affected)
```

The `WHERE` clause filtered the table down to 1492 qualifying rows, which, when grouped, produced the 164 rows of output.

To run the tests, follow the same scheme as described in Listing 5.3, but using the new query from Listing 5.4. The results are The relative effort required to execute the query against each indexing scheme is shown in Table 5.2.

Run 1: No Nonclustered Index	Table 'SalesOrderDetail'. Scan count 1, logical reads 1238. Non read activity: 10%.
Run 2: With Index – No Included Columns	Table 'SalesOrderDetail'. Scan count 1, logical reads 1238. Non read activity: 10%.
Run 3: With Included Columns	Table 'SalesOrderDetail'. Scan count 1, logical reads 761. Non read activity: 8%.

Table 2: Results of running the second query three times with different nonclustered indexes available

Both the first and second test resulted in the same plan: a complete scan of `theSalesOrderDetail` table. For reasons that were covered in detail in Level 4, the `WHERE` clause was not sufficiently selective to benefit from a non-covering index. Also, the rows comprising any one group are scattered throughout the table. As the table was being read, each row had to be matched to its group; and operation that consumes processor time and memory.

The third test found everything that it needed in the nonclustered index; but, unlike the previous query, it did not find the rows located contiguously within the index. he rows that comprise each individual group are contiguous within the index; but the groups themselves are scattered over the length of the index. Therefore, SQL Server scanned the index.

Scanning the index instead of the table had two advantages:

- The index is smaller than the table, requiring less reads.