SQL ServerCentral.com
Proactively monitor your growing SQL Server environment.
Try free SQL Monitor
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: G+ 0
Rate this | Join the discussion | Add to briefcase

Filtered Indexes in SQL Server: Stairway to SQL Server Indexes Level 7

By David Durant, 2011/08/24

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 several preceding levels, we have said there is an exception to the rule that every row of the table generates an entry in an index. Some indexes have fewer entries than the corresponding table has rows. These indexes are called **filtered indexes**; a feature that was introduced with SQL Server 2008.

A description of the tables used in the examples in this level, from the *AdventureWorks* database, can be found in the *Resources* section at the end of the article.

Filtering an Index

You filter an index the same way you filter a `SELECT` statement; with a `WHERE` clause; as in:

```
IF EXISTS ( SELECT   *
            FROM     sys.indexes
            WHERE    OBJECT_ID = OBJECT_ID('
                     AND name = 'FI_Specialo
    DROP INDEX Sales.SalesOrderDetail.FI_Sp
GO

CREATE INDEX FI_SpecialOfferID
ON Sales.SalesOrderDetail (SpecialOfferID)
WHERE SpecialOfferID;
```

Related Articles
FORUM Filter index
2010/05/07
Filter index

ARTICLE Strange Filtered Index Problem
2015/08/07
What I thought was a strange filtered index problem but turned out to be a DB settings problem

BLOG SQL Server Filtered Index

BLOG

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

licy. Terms of Use. Report Abuse.

Listing 7.1: Creating a filtered index

The primary reason for filtering an index is to eliminate one or more highly unselective values from the index. Consider the `SpecialOfferID` column of the `SalesOrderDetail` table. The 121,317 rows contain twelve different `SpecialOfferID` values, ranging from 1 to 15. The number of rows for each value is shown in table 7.1.

```
SpecialOfferID RowCount
-------------- -----------

1              115884
2              3428
3              606
13             524
14             244
16             169
7              137
8              98
11             84
4              80
9              61
5              2
```

Table 7.1: RowCount summary for the SpecialOfferID colum of the SalesOrderDetail table

The vast majority of the rows, over 95%, have a `SpecialOfferID` value of 1. Therefore, a nonclustered index on the `SpecialOfferID` column would not benefit a query that requested rows `WHERE SpecialOfferID = 1`. That query would use a table scan to find the 115,884 requested rows. However, the index *would* benefit queries seeking rows `WHERE SpecialOfferID = 5`.

The `CREATE INDEX` statement shown in Listing 7.1 generates an index that has no entries for the 115,884 `SpecialOfferID = 1` rows: entries that would be of no value anyway. Thus, the resulting index is small, light, and efficient, containing just 5433 entries.

In our `SalesOrderDetail` example, the dominant value that needed be filtered out was '1'. In your own applications, the most common dominant value is probably `NULL`. In a typical transactional database, there are nullable columns in which nulls predominate and `NOT NULL` values are the exception. When creating an index on these columns, always consider filtering out the nulls.

To verify the benefit of filtered indexes, we'll run the query shown in Listing 7.2 six times:

- Three times against an unfiltered index on the `SpecialOfferID` column, with parameter values of '1', '13' and '14'
- Three times against an filtered index on the `SpecialOfferID` column, with the same parameter values.

As can be seen from the **RowCount** summary shown in Table 7.1, our first parameter, '1', occurs in 95% of the rows; the

second, '13', in 4% of the rows, and the last, '14', in 2% of the rows.

```
SELECT   *
FROM     Sales.SalesOrderDetail
WHERE    SpecialOfferID = <</span>parameter
ORDER BY SpecialOfferID ;
```

Listing 7.2: A query to test our filtered index

As usual, we use "reads" as our primary performance metric; and we use SQL Server Management Studio's "Show Actual Execution Plan" to observe SQL Server's plan for each query. We begin by creating an unfiltered index, as shown in Listing 7.3.

```
SELECT   *
FROM     Sales.SalesOrderDetail
WHERE    SpecialOfferID = &lt;parameter valu
ORDER BY SpecialOfferID ;
```

Listing 7.3: Creating the unfiltered index

We execute the query once for each of our three parameter values, and then recreate the index; this time as a filtered index, as shown in Listing 7.1. This filtered index will contain 1/20th (5%) the number of entries of the unfiltered index. Again, we execute the query once for each of our three parameter values. The combined results are shown in Table 7.2.

WITH UNFILTERED INDEX:		
Parameter Value	**Reads**	**Plan**
1	1238	Table scan
13	1238	Table scan
14	759	Retrieve bookmark values from index. Use them to retrieve rows from table
WITH FILTERED INDEX:		
Parameter Value	**Reads**	**Plan**
1	1238	Table scan
13	1238	Table scan
14	759	Retrieve bookmark values from index. Use them to retrieve rows from table

Table 7.2: Results running the query with both filtered and unfiltered index

As the table 7.2 shows, the results are the same regardless of which index is used. In other words, the filtered index is just as beneficial as the unfiltered index, which is twenty times larger. We get a tremendous saving in disk space at no cost in query performance.

In the sample index that we just examined, the index key column(s) and the filtering column(s) were the same column. Although this is often the case, it is not a requirement. When we specified our filtering `WHERE` clause, we told SQL Server: "If you are looking for rows whose `SpecialOfferID value is <> 1`, this index has the entries for those rows." It can be beneficial for SQL Server to know this information, regardless of the index key.

Consider the index that we have created in previous levels to help the warehouse staff search the `SalesOrderDetail` table for product related information. The last version of the index, the one appearing in Levels 5 & 6, is repeated in Listing 7.4.

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

Listing 7.4: Creating the nonclustered index with included columns

The resulting index would contain the following index rows:

```
:- Search 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
```

If the warehouse frequently requests product information for rows whose special offer category is not "1", and seldom requests information for category 1 rows, then adding a `WHERESpecialOfferID != 1` clause to the create index statement makes sense. The end result is a much smaller index that covers the majority of the requests. So, let's we modify our `CREATE INDEX` statement as demonstrated in Listing 7.5.

```
CREATE NONCLUSTERED INDEX FK_ProductID_Modi
  ON Sales.SalesOrderDetail (ProductID,Modi
  INCLUDE (OrderQty,UnitPrice,LineTotal)
  WHERE SpecialOfferID <>1
```

Listing 7.5: Creating the filtered, nonclustered index with included columns

Next, execute the query shown in Listing 7.6

```
SELECT  ProductID ,
        ModifiedDate ,
        SUM(OrderQty) 'No of Items' ,
        AVG(UnitPrice) 'Avg Price' ,
        SUM(LineTotal) 'Total Value'
FROM    Sales.SalesOrderDetail
WHERE   SpecialOfferID &lt;&gt; 1
GROUP BY ProductID ,
        ModifiedDate
```

Listing 7.6: Querying the filtered, nonclustered index with included columns

SQL Server Management Studio informs us that the filtered index was scanned and that 35 pages (possible spread over as few as 5 extents) were read to generate the 2,102 rows of output.

There are a couple of important issues to bear in mind when deciding on your strategy for designing and using filtered indexes.

You might be surprised to learn that changing the `WHERE` clause in the previous statement from "`SpecialOfferID <> 1`" to "`SpecialOfferID = 2`" will prevent SQL Server from using the filtered index. This is because SQL Server compares the `WHERE` clause of the `SELECT` statement against the `WHERE` clause of the `CREATE INDEX` statement for *lexical* equivalence, not for *logical* equivalence. Therefore, SQL Server does not realize that the filtered index covers the query.

In addition, you could not cause SQL Server to use the filtered index by using a redundant where clause that combined both criteria, like so: "`WHERE SpecialOfferID <> 1 AND SpecialOfferID = 2`". In a later level we cover *Index Hints*, which give you the ability to influence SQL Server's choice of index. For now, however, just remember that SQL Server is making a lexical decision when evaluating a filtered index.

When creating a filtered index, verify that you are not creating that index to compensate for a failure of third normal form in your database design.

NOTE:
We are about go beyond the normal scope of this Stairway series, in order to make some brief comments about logical database design, comments that are relevant to filtered indexes which may help you avoid a common design flaw. These comments that will be presented without background or elaboration.

Most commonly, a failure of third normal form is caused by a failure to recognize **entity subtypes** in your design. Consider the representation of the contents of the `Products` table, shown in Figure 7.1.

ProductID	Description	Type	Price	Author	IssuesPerYear
{Primary Key}					
44E	Roots	Book	44.50	Alex Haley	
17J	Time	Periodical	18.00		52
22D	Gift from the Sea	Book	27.00	Anne Morrow Lindbergh	
18K	National Geographic	Periodical	48.00		12
7BK	Good Housekeeping	Periodical	27.00		12

Figure 7.1: The contents of a Products table containing two subtypes

It is apparent from the data in this table that there are two subtypes of products: books and periodicals. Only books have author information and only periodicals have issues-per-year information. The proper way to model subtypes is to have one table that holds the information common to all types, plus one table for each subtype. All tables have the same primary key

column(s), with the primary key of the subtype tables also being the foreign key that links them to the main table.

Thus, the `Products` table would best be broken into the following three separate tables, as shown in Figures 7.2, Figure 7.3 and Figure 7.4.

Products Table

ProductID	Description	Price
(Primary Key)		
44E	Roots	14.50
17J	Time	13.00
22D	Gift from the Sea	37.00
18K	National Geography	33.00
79K	Good Housekeeping	37.00

Table 7.2: A Products table containing information common to all products

Books table

ProductID	Author
(Primary Key and Foreign Key)	
44E	Alex Haley
22D	Anne Morrow Lindbergh

Table 7.3: A Books table containing information relevant only to book products

Periodicals table

ProductID	IssuesPerYear
(Primary Key and Foreign Key)	
17J	52
18K	12
79K	12

Table 7.4: A Periodicals table containing information relevant only to periodical products

As such, filtered indexes, especially ones that filter out `NULL` values, could be an indication that an entity subtype has been overlooked.

The correct solution is to redefine the tables, but many might question whether it not is really matters if the complete information for a subtype is in two easily-joined tables or in one filtered index. It does. Application developers and application development tools know nothing of your indexes; they can only see your tables. If the structure of your tables does not reflect the structure of the business, developers will struggle to build and maintain an application on top of your database.

A filtered index eliminates unusable entries from the index, producing an index that is as beneficial as, but much smaller than, an unfiltered index. An index is filtered by specifying a `WHERE` clause within the `CREATE INDEX` statement. The columns specified in the `WHERE` clause can be different from the columns specified for the index key, or from the columns specified in the `INCLUDE` clause.

If a certain subset of a table's data is frequently requested, a filtered index can also be a covering index, resulting in a considerable reduction in IO.