Filtered Indexes Work-Around

  By [Michelle Ufford](http://sqlfool.com/author/michelle/) | [February 1, 2010](http://sqlfool.com/2010/02/filtered-indexes-work-around/)| [Performance & Tuning](http://sqlfool.com/category/performance-tuning/), [SQL Server](http://sqlfool.com/category/sqlserver/), [SQL Tips](http://sqlfool.com/category/tips-suggestions/), [Syndication](http://sqlfool.com/category/syndication/)

0  
[inShare](javascript:void(0);)

Recently, I needed to create a stored procedure that queried a rather large table. The table has a filtered index on a date column, and it covers the query. However, the Query Optimizer was not using the index, which was increasing the execution time (not to mention IO!) by at least 10x. This wasn’t the first time I’ve had the Optimizer fail to use a filtered index. Normally when this happens, I use a table hint to force the filtered index — after I verify that it is indeed faster, of course. However, since this was a stored procedure, I was receiving the following error message whenever I tried to execute the proc:

Query processor could not produce a query plan because of the hints defined in this query. Resubmit the query without specifying any hints and without using SET FORCEPLAN.

SQL Server would not allow me to execute the stored procedure using the filtered index hint. If I removed the hint, it executed, but it used a different, non-covering and far more expensive index. For those of you not familiar with this issue, allow me to illustrate the problem.

First, create a table to play with and populate it with some bogus data:

|  |
| --- |
| Create Table dbo.filteredIndexTest  (  myID int Identity(1,3)  , myDate smalldatetime  , myData char(100)    Constraint PK\_filteredIndexTest  Primary Key Clustered(myID)  );    Set NoCount On;  Declare @date smalldatetime = '2010-01-01';    While @date < '2010-02-01'  Begin    Insert Into dbo.filteredIndexTest  (  myDate  , myData  )  Select @date  , 'Date: ' + Convert(varchar(20), @date, 102);    Set @date = DateAdd(minute, 1, @date);    End;    Select Count(\*) From dbo.filteredIndexTest; |

It looks like this will generate 44,640 rows of test data… plenty enough for our purposes. Now, let’s create our filtered index and write a query that will use it:

|  |
| --- |
| Create NonClustered Index IX\_filteredIndexTest\_1  On dbo.filteredIndexTest(myDate)  Include (myData)  Where myDate >= '2010-01-27';    Select Distinct myData  From dbo.filteredIndexTest  Where myDate >= '2010-01-28'; |

If you look at the execution plan for this query, you’ll notice that the Optimizer is using the filtered index. Perfect! Now let’s parameterize it.

|  |
| --- |
| Declare @myDate1 smalldatetime = '2010-01-28';    Select Distinct myData  From dbo.filteredIndexTest  Where myDate >= @myDate1; |

Uh oh. Looking at the execution plan, we see that SQL Server is no longer using the filtered index. Instead, it’s scanning the clustered index! Why is this? There’s actually a good explanation for it. The reason is that I could, in theory, pass a date to my parameter that fell outside of the filtered date range. If that’s the case, then SQL Server could not utilize the filtered index. Personally, I think it’s a bug and SQL Server should identify whether or not a filtered index could be used based on the actual value submitted, but… that’s a whole other blog post. :)

So what can we do? Well, dynamic SQL may be able to help us out in this case. Let’s give it a go. First, let’s try parameterized dynamic SQL.

|  |
| --- |
| Declare @mySQL1 nvarchar(2000)  , @myParam nvarchar(2000) = '@p\_myDate2 smalldatetime'  , @myDate2 smalldatetime = '2010-01-28';    Set @mySQL1 = 'Select Distinct myData  From dbo.filteredIndexTest  Where myDate >= @p\_myDate2';    Execute sp\_executeSQL @mySQL1, @myParam, @p\_myDate2 = @myDate2; |

Looking at the execution plan, we see we’re still scanning on the clustered index. This is because the parameterized dynamic SQL resolves to be the exact same query as the one above it. Let’s try unparameterized SQL instead:

|  |
| --- |
| Declare @mySQL2 nvarchar(2000)  , @myDate3 smalldatetime = '2010-01-28';    Set @mySQL2 = 'Select Distinct myData  From dbo.filteredIndexTest  Where myDate >= ''' + Cast(@myDate3 As varchar(20)) + '''';    Execute sp\_executeSQL @mySQL2;    -- Drop Table dbo.filteredIndexTest; |

Voila! We have a seek on our filtered index. Why? Because the statement resolves to be identical to our first query, where we hard-coded the date value in the WHERE clause.

Now, I want to stress this fact: you should always, ALWAYS use parameterized dynamic SQL whenever possible. Not only is it safer, but it’s also faster, because it can reuse cached plans. But sometimes you just cannot accomplish the same tasks with it. This is one of those times. If you do end up needing to use unparameterized dynamic SQL as a work-around, please make sure you’re validating your input, especially if you’re interfacing with any sort of external source.

There’s an even easier work-around for this problem that [Dave](http://twitter.com/CrappyCodingGuy) ([http://www.crappycoding.com](http://www.crappycoding.com/)) shared with me: [recompile](http://msdn.microsoft.com/en-us/library/ms181714.aspx).

Adding “Option (Recompile)” to the end of your statements will force the Optimizer to re-evaluate which index will best meet the needs of your query every time the statement is executed. More importantly, it evaluates the plan based on the actual values passed to the parameter… just like in our hard-coded and dynamic SQL examples. Let’s see it in action:

|  |
| --- |
| DECLARE @myDate4 SMALLDATETIME = '2010-01-28';    SELECT DISTINCT myData  FROM dbo.filteredIndexTest  WHERE myDate >= @myDate4  OPTION (RECOMPILE);    DECLARE @myDate5 SMALLDATETIME = '2010-01-20';    SELECT DISTINCT myData  FROM dbo.filteredIndexTest  WHERE myDate >= @myDate5  OPTION (RECOMPILE); |

If we look at the execution plans for the 2 queries above, we see that the first query seeks on the filtered index, and the second query scans on the clustered index. This is because the second query cannot be satisfied with the filtered index because we initially limited our index to dates greater than or equal to 1/27/2010.

There are, of course, trade-offs associated with each approach, so use whichever one best meets your needs. Do you have another work-around for this issue? If so, please let me know. :)

Update:

Alex Kuznetsov (<http://www.simple-talk.com/author/alex-kuznetsov/>) shared this method too:

|  |
| --- |
| DECLARE @myDate1 SMALLDATETIME = '2010-01-28';    SELECT DISTINCT myData  FROM dbo.filteredIndexTest  WHERE myDate = @myDate1  AND myDate >= '2010-01-27'; |

Like the other examples, this will result in an index seek on the filtered index. Basically, by explicitly declaring the start date of your filter, you’re letting the Optimizer know that the filtered index can satisfy the request, regardless of the parameter value passed. Thanks for the tip, Alex! :)