

## Deleting Unused or Expensive Indexes

29 Jun 2010 7:49 PM | [Q](#)

In my earlier posts, I illustrated two ways to identify some missing indexes by using SQL Server 2005 Performance Dashboard and Database Engine Tuning Advisor. In this post, I will look at the other side of the coin and delete indexes that are unused or that cost more resources than they save.

In my last post, I recommended adding the recommended indexes until your Windchill installation is within 5 percent of the maximum performance that Database Engine Tuning Advisor identifies. Getting the last 5 percent usually means adding so many indexes that index maintenance consumes so many resources that you lose performance.

In this post, I will use Transact-SQL statements to identify indexes that are expensive or unused. This detection of expensive indexes uses sampling. You want to have a significant sample of data to perform this on. If your sample is very sparse, you might falsely conclude that some indexes are unused simply because your sample failed to include appropriate queries.

Also remember that Windchill usage patterns might change when contracts finish or when new projects start, resulting in new indexes being needed and old indexes becoming unused (but still needing to be maintained). Because of this usage pattern impact, I recommend repeating the process of identifying indexes to add and to delete every quarter to keep Windchill functioning well.

In this post, I will be looking at doing tasks in SQL Server Management Studio by using Transact-SQL statements and not by using tools like in my prior posts. In other words, I am going deep and bordering on the murky territory of developers.

### Finding Unused Indexes

The following Transact-SQL will return the indexes (with associated tables) that are likely candidates for deletion.

```
use [wcAdmin]
go
SELECT TableName = OBJECT_NAME(s.[object_id])
       ,IndexName = i.name
       ,user_updates
       ,system_updates
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
      AND s.index_id = i.index_id
WHERE  OBJECTPROPERTY(s.[object_id], 'IsMsShipped') = 0 -- Only wcadmin defined.
      AND user_seeks = 0
      AND user_scans = 0
      AND user_lookups = 0
      AND i.name IS NOT NULL -- Ignore HEAP indexes.
ORDER BY user_updates DESC
```

Note the following important facts:

- The information is based on the data since SQL Server started. (This is your critical sample.)
- If a table is not modified (updated), it will not be listed. To demonstrate this:
  1. Run the query.
  2. Stop SQL Server.
  3. Restart SQL Server.
  4. Rerun the query.
- You will get no records.
- Run this code after SQL Server has at least one week of data.
- **Warning:** If your server has automatic updates enabled, add "Analysis Monday Evening" to your calendar, just before "Patch Tuesday." (Microsoft updates are typically released on Tuesday.)

You may want to create a persistent table and record the above information every week to this table (using SQL Server Agent to execute it automatically). This will increase your sample size, but it also means that you would need to filter this table. To do this, follow these steps:

1. Create your persistent table.

```
CREATE TABLE [dbo].[PerformanceUnusedIndexes] (
  [RecordedAt] DateTime default (getdate()),
  [TableName] [nvarchar](128) NULL,
  [IndexName] [sysname] NULL,
  [user_updates] [bigint] NOT NULL,
  [system_updates] [bigint] NOT NULL
)
```

2. Do not use the **wcadmin** schema. Separate your ad hoc table from the tables that PTC creates.

3. Schedule the following code to execute regularly with SQL Server Agent.

```
use [wcadmin]
go
INSERT INTO [PerformanceUnusedIndexes]
  ([TableName]
  , [IndexName]
  , [user_updates]
  , [system_updates])
SELECT TableName = OBJECT_NAME(s.[object_id])
```

```

,IndexName = i.name
,user_updates
,system_updates
FROM sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
AND s.index_id = i.index_id
WHERE OBJECTPROPERTY(s.[object_id], 'IsMsShipped') = 0 -- Ignore HEAP indexes.
AND user_seeks = 0
AND user_scans = 0
AND user_lookups = 0
AND i.name IS NOT NULL -- Ignore HEAP indexes.
ORDER BY user_updates DESC

```

4. Run the following query to identify the indexes that are unused indexes across all the collected sample data.

```

Select [TableName] ,
[IndexName],
Sum([user_updates]),
Sum([system_updates])
From [PerformanceUnusedIndexes]
Group by [TableName] ,
[IndexName]
Having Count(1)=(Select MAX(cnt) From
(Select COUNT(1) as cnt From [PerformanceUnusedIndexes]
Group by [TableName] ,
[IndexName]) MaxCount)

```

### Finding Expensive Indexes

We can do the same process by using the following bit of Transact-SQL. We exclude unused indexes from the list so that we can get a payback datum, [Payback], on the cost of maintaining the indexes.

```

SELECT (user_updates + system_updates) as [MaintenanceCost]
,(user_seeks + user_scans + user_lookups) as [FrequencyOfUse]
,cast(user_updates + system_updates as decimal)
/cast(user_seeks + user_scans + user_lookups as decimal) as [Payback]
,TableName = OBJECT_NAME(s.[object_id])
,IndexName = i.name
FROM sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
AND s.index_id = i.index_id
WHERE OBJECTPROPERTY(s.[object_id], 'IsMsShipped') = 0 -- wcAdmin installed indexes.
AND (user_updates + system_updates) > 0 -- Only those with a cost.
and (user_seeks + user_scans + user_lookups) > 0 --Only those with usage.
AND i.name IS NOT NULL -- skip HEAP.
ORDER BY cast(user_updates + system_updates as decimal)
/cast(user_seeks + user_scans + user_lookups as decimal) DESC

```

An example of output is as follows.

	MaintenanceCost	FrequencyOfUse	Payback	TableName	IndexName
1	2	2	1.000000000000000000	NotificationList	NotificationList\$UNIQUE
2	2	2	1.000000000000000000	NotificationList	PK_NotificationList
3	1	2	0.500000000000000000	PolicyAcl	PolicyAcl\$UNIQUE
4	1	2	0.500000000000000000	PolicyAcl	PK_PolicyAcl
5	3	7	0.4285714285714285714	RecentUpdate	PK_RecentUpdate
6	137	394	0.3477157360406091370	ScheduleQueue	PK_ScheduleQueue
7	2	6	0.3333333333333333333	ManagedBaseline	PK_ManagedBaseline
8	1	4	0.2500000000000000000	PDMLinkProduct	PK_PDMLinkProduct
9	6	884	0.0067873303167420814	ScheduleQueueEntry	ScheduleQueueEntry\$COMPOSITE
10	6	1107	0.0054200542005420054	ScheduleQueueEntry	PK_ScheduleQueueEntry

In general, you are interested in looking at items that have a payback of 0.5 or higher. Do not delete these. We need to do some analysis first.

The next step would be to do a count on the number of rows in each table. A table with a low row count (< 1,000) has little impact from the presence or omission of an index (other than a primary key). A table with 10,000,000 rows needs an index for good performance.

You should never delete a primary key index. These rows are actually easy to exclude from your delete list. Just add "AND i.is\_primary\_key=0" to the Transact-SQL. Alternatively, the index name might follow [Hungarian notation](#) and start with "PK\_" (hope that the name is truthful in its implication).

You should look at the definition of each index (and other indexes on the same table). On occasion, they might be minor variations of each other (especially those that Database Engine Tuning Advisor adds), effectively a redundant index. Try to rewrite redundant indexes into a single index.

For example, look at the following two indexes.

```

localhost\Win...SQLQuery4.sql* localhost\Win...SQLQuery3.sql* localhost\Win...SQLQuery2.sql* localhost\Win...SQLQuery1.sql* Objec
** Object: Index [ScheduleQueueEntry$COMPOSITE] Script Date: 04/18/2010 08:35:51 ***/
CREATE NONCLUSTERED INDEX [ScheduleQueueEntry$COMPOSITE] ON [wcadmin].[ScheduleQueueEntry]
(
    [idA3A5] ASC,
    [codeC5] ASC,
    [entryNumber] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY =
USE [wcadmin]
GO
/***** Object: Index [ScheduleQueueEntry$COMPOSITE1] Script Date: 04/18/2010 08:36:06 ***
CREATE NONCLUSTERED INDEX [ScheduleQueueEntry$COMPOSITE1] ON [wcadmin].[ScheduleQueueEntry]
(
    [entryNumber] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY =

```

Dropping these two indexes and creating one that is **[entryNumber],[idA3A5],[codeC5]** would clearly eliminate the need for the second index. If the Database Engine Tuning Advisor analysis later suggests the **[idA3A5],[codeC5],[entryNumber]** index, you should likely revert to the original index because maintaining both **[idA3A5],[codeC5],[entryNumber]** and **[entryNumber],[idA3A5],[codeC5]** is more expensive than the original index. (It will take more disk space at a minimum.) If you rework indexes, make sure that you keep the original versions so that you can easily revert later if necessary.

## Changing Primary Keys

This is hacker-level trick (it is not recommended, without warranty). If a primary key shows up as a possible candidate, you may want to see if there is another unique index on the same table (typically, an alternative primary key) that is being used. Then, you change this alternative primary key to the primary key and drop the expensive primary key index.

In the previous example, you can see that **PolicyAcl** has two indexes, and you want to see if there are other indexes that you can use as a primary key. This is easy to do with the following Transact-SQL.

```

SELECT distinct OBJECT_NAME(s.[object_id]) As TableName
    ,i.name as IndexName
    ,is_unique_constraint
    ,is_unique
    ,is_primary_key
FROM sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
WHERE OBJECTPROPERTY(s.[object_id], 'IsMsShipped') = 0 -- wcAdmin installed indexes.
    AND 'PolicyAcl' = OBJECT_NAME(s.[object_id])

```

	TableName	IndexName	is_unique_constraint	is_unique	is_primary_key
1	PolicyAcl	PK_PolicyAcl	0	1	1
2	PolicyAcl	PolicyAcl\$UNIQUE	0	1	0

Therefore, there are no better primary keys available. Typically, it is rare to find a viable alternative key with well-engineered databases like Windchill, but it can occur with atypical load patterns.

## Summary of Index Tuning

In this series of posts, I described two ways of adding missing indexes by using the following tools:

- SQL Server 2005 Performance Dashboard
- Database Engine Tuning Advisor

I also provided some guidance for when to leave good enough alone. (The last 5 percent of performance is often counterproductive.) Last, I discussed identifying and deleting unused and expensive indexes, including identifying redundant indexes.

The process of index tuning is an ongoing process that changes over time. You need to revisit indexes on a regular basis.

---

Ken Lassen is part of the original team that created Dr. GUI of MSDN and specializes in new and resurrected commercial product architecture. He developed architecture for several Microsoft websites, including the original [MSDN](#) site and the current [Microsoft Partner Network](#) site. He's equally at home with SQL Server, XHTML, Section 508 accessibility standards, globalization, Security Content Automation Protocol (SCAP) security, C#, and ASP.NET server controls. When he is not having fun with technology, he enjoys taking lunch-break hikes in the North Cascades.