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

redgate | Hub
Insights
Learning
Forums
Ideas
Discover the new Redgate Hub ›

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: G+ 1
Rate this | Join the discussion | Add to briefcase

# SQL Server Indexing Best Practices: Stairway to SQL Server Indexes Level 15

By David Durant, 2012/03/28

## 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 this article we make recommendations: fourteen guidelines drawn from information presented throughout this Stairway. These guidelines will help you create the best indexing scheme for your database.

The format was taken from *Framework Design Guidelines;* Krzysztof Cwalina and Brad Abrams' excellent work on standards for .Net application development, published by Addison Wesley. Each recommendation is specified by the word(s) **do, consider, avoid**, and **do not**, which have the following meanings.

- Do: guidelines should always be followed.
- Consider: guidelines should generally be followed; but if you fully understand both the reason behind the guideline and your reason for not following it; then you should feel free to deviate from the guideline.
- Avoid: is the opposite of consider. In general the guidelines suggest something that should not be done, but if you completely understand why it should not be done, and you understand your reason for doing it anyway; then do it.
- Do not is the stronger version of avoid, and indicates something that should never be done. Like do guidelines, do not guidelines should always be followed.

Related Articles

ARTICLE Index Creation Guidelines
2007/10/02
Index Creation Guidelines for SQL Server can be pretty sparse. Usually there are a couple, clustered...

FORUM basic guidelines on creating Indexes
2009/03/27
basic guidelines on creating Indexes

FORUM
FORUM
FORUM

Tags

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

Write For Us
Find out more

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

The primary purpose of an index is to improve the performance of your application's data gathering and data manipulation operations. Until you know what those operations are, you have no hope of improving them.

It's nice to be involved in the application from the beginning; participating in the design and development. But this is the exception not the rule. If you are inheriting an already implemented database and application, take two approaches to understanding what you have inherited, the outside approach and the inside approach.

The outside approach consists of learning from the users, talking with them, watching them use the application, reading any user oriented documentation and instructions, and reviewing current forms and reports.

The inside approach involves examining the application itself: both the definition of the application and the execution of the application. Tools such as Activity Monitor, Profiler, the *sys.dm_db_index usage_stats* dynamic management view, and the *sys.dm_db_missing_index_XXX* family of dynamic management views provide information about popular queries, long running queries, popular indexes, unused indexes, and indexes that should exist but do not.

Examining the source location of popular or poorly performing queries, such as Reporting Services templates, T-SQL job steps, T-SQL tasks in SSIS applications, and stored procedures can yield information about why the statements are important to the application and, therefore, should be well optimized.

Armed with this information, you can make good decisions regarding which indexes are beneficial, and which are not.

Too many indexes are as bad a problem as too few indexes. There is no magic 'best number of indexes' for a table. Every table is different. However, once you have indexed the primary key, any candidate keys, and the appropriate foreign keys, any other potential index requires careful analysis on your part before you add it to your database.

Whether it is daytime processing versus off-hours processing, or transactional processing run against an OLTP database versus report processing run against a replicated copy of that same database, different situations require different indexing.

A database that receives massive infusions of new data each night should have fewer indexes during that infusion than during normal processing hours. A volatile database with limited querying requirements needs fewer indexes than an *updated-once-each-night* reporting database that handles complex report generation.

Although primary keys are not required by SQL Server, a table without a primary key is dangerous thing in a transactional or reporting database, as its rows are not guaranteed to be unique. If duplicate rows are allowed, they will happen, and you will never know whether the same instance of a subject was entered twice, or whether

you have separate instances with insufficient information to distinguish one from the other.

Although not required by SQL Server, primary keys are a cornerstone of relation theory, the basic building block of all relational systems. Without primary key constraints, and their associated UNIQUE indexes, relational operations will yield unexpected results and poor performance.

In addition, many client side development tools and components need your tables to have primary keys. For example, both the ADO.Net SqlCommandBuilder component and Visual Studio's Entity Data Modeler depend upon targeted database tables having primary key constraints. Remember that the name of the primary key constraint becomes the name of the index that is automatically created to enforce that constraint.

This Stairway's *Level 3 – Clustered Indexes* covered the benefits of having a clustered index on a table; that is, of having the table be a clustered index rather than be a heap. The main benefit is the simple fact that the user community as a whole tends to view a table's data in a certain default sequence; hence the advantage of maintaining the rows in that sequence.

If you follow these recommendations laid out in this Level, every table will have a primary key. Therefore, every table will have at least one index, and probably more. Thus, making one of those indexes the clustered index will not increase the number of indexes, but it will just give your table a better structure than that of a heap.

When deciding on the clustered index key, remember the guidelines specified in *Level 6 – Bookmarks*: a clustered index key should be unique, short, and non-volatile.

Using a foreign key as the left most column(s) of the clustered index key will cluster child information around a common parent; which is a typical processing requirement. Your credit card charges are associated with your card, my charges are associated with my card. That relationship is stronger that the one that associates a charge with the merchant who sold the item, or with the financial institution that is processing the charge. Card number is the foreign key that belongs in the Charge table's clustered index's key; not merchant number or bank number. By making card number the left most column of the clustered index, all charges for a single cardholder will be clustered together on the same data page(s).

Add additional non-volatile column(s) to the key, as necessary, to ensure uniqueness of the clustered index key.

Using a foreign key as the left most column(s) of the clustered index key will cluster child information around a common parent; which is a typical processing requirement. Your credit card charges are associated with your card, my charges are associated with my card. That relationship is stronger that the one that associates a charge with the merchant who sold the item, or with the financial institution that is processing the charge. Card number is the foreign key that belongs in the Charge table's clustered index's key; not merchant number or bank number. By making card number the left most column of the clustered index, all charges for a single cardholder will be clustered together on the same data page(s).

Add additional non-volatile column(s) to the key, as necessary, to ensure uniqueness of the clustered index key.

The old cliché is "Never index the Gender column". A typical page of the table will have half female rows and half male rows, and will be accessed whether the request is for female rows or for male rows. A table scan will always be the best decision for any WHERE `GENDER` = ... query; therefore, such an index will never be of benefit to the optimizer.

If a large percentage of the rows have the same value for a particular column, or are null, create a filtered index that on that column. Those queries searching for rows of the rarer values will use this small, efficient index; queries looking for the rows of the common value will do a table scan. And SQL Server can easily determine which is which.

If a relatively new clustered index table contains one month's worth of rows and will be allowed to grow until it contains one year's worth of rows, rebuild the index with a FILLFACTOR value of 7 or 8 percent. This will cause the table to consume the same number of pages now as a year from now. Potential problems with space requirements and processing performance, such as the number of I/Os required for a table scan, will appear sooner rather than later.

If a table has already reached its planned maximum size, the previous guideline does not apply. In this case, the fill factor that will result from the table's normal activity should be specified to begin with. As was mentioned in *Level II – Index Fragmentation*, for a typical transaction table that has continual inserting but only periodic deleting, this

value is 75. For a table that has equal amounts of insert and delete activity, 90 to 95 would be a good choice.

A corollary to this guideline is: **Do** drop a table's nonclustered indexes before dropping its clustered index. Doing otherwise will cause the nonclustered indexes to be rebuilt unnecessarily. Transitioning a table between being a heap and being a clustered index always causes the table's non-clustered indexes to be rebuilt because the contents of the bookmarks must change from row identifiers to clustered index keys.

If an indexed is frequently scanned, then, as mentioned in *Level 11 – Index Fragmentation*, the external fragmentation of that index is important; for it has a major impact on the effort required to scan all or some of the leaf level of an index. If this is the case, consider reorganizing the index when external fragmentation reaches ten percent, and consider rebuilding the index when external fragmentation reaches thirty percent. For most transactional environments, the values mentioned above represent the point at which the benefit of performing the reorganization or rebuild of the index outweighs the cost of doing it.

However, if an index is being searched for a specific key value, then external fragmentation has little or no impact on performance. The IO required to traverse one page at each level from the root page to the leaf level, will be the same regardless of external fragmentation. In this situation, reorganizing or rebuilding the index will have little impact on performance.

The key word here is "regular", because only by knowing what your applications are doing can you determine when statistics need to be updated. *Level 14 – Index Statistics* illustrates why some statistics become outdated faster than others.

These guidelines evolved from the experiences of many developers working with SQL Server for many years in a variety of environments. Following them can help you create the best possible indexing for your database.

**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: G+ 1

Rate this | Join the discussion | Add to briefcase

Total article views: 17154 | Views in the last 30 days: 49