Learn SQL Server Administration in a Month of Lunches

DON JONES

MANNING
brief contents

1 ▪ Before you begin  1
2 ▪ Server assessment and configuration  8
3 ▪ T-SQL crash course  16
4 ▪ Managing databases  34
5 ▪ Backup and recovery  44
6 ▪ Authentication: who are you?  56
7 ▪ Authorization: what are you allowed to do?  65
8 ▪ Accounting: what did you do?  75
9 ▪ Analyzing indexes  82
10 ▪ Maintaining indexes  92
11 ▪ Tuning index designs  101
12 ▪ Reading query execution plans  110
13 ▪ Block and deadlock analysis  119
14 ▪ Automating management with SQL Server Agent  128
15 ▪ Multiserver management  137
16 ▪ Windows PowerShell and SQL Server  143
17 ▪ Using Extended Events  153
18 ■ Monitoring and analyzing performance 166
19 ■ Options for high availability 178
20 ■ Virtualizing SQL Server 189
21 ■ Moving, migrating, and upgrading databases 197
22 ■ SQL Server performance checklist 205
23 ■ Never the end 207
In the previous chapter, I explained how SQL Server indexes are built, and broadly hinted at how SQL Server uses them. Indexes will play a big part in many upcoming chapters, as they’re one of the biggest performance-related items that an administrator can potentially play with. In keeping with this book’s overall theme, though, I’m assuming that your databases are already indexed using whatever indexes the database application developer figured would work well.

But indexes, like cars, don’t work very well, for very long, unless they’re properly maintained.

### 10.1 Understanding index fragmentation

The biggest problem—the only operational problem—that indexes run into is fragmentation. Remember that an index is an ordered reference to a particular table’s data. The index takes all the values from a designated key column, like customer name or product stock number, and lists those values in a specific order. The idea is that it’s much easier to find a piece of data when it’s already sorted: you can seek to the section of the index you need, and find the item you want. That’s what an index’s B-tree structure helps to facilitate.

**NOTE** I use the word seek specifically because it’s the term SQL Server uses to describe a quick lookup in a B-tree. Another term is scan, wherein SQL Server literally starts at the top of the list and reads its way through, one item at a time. Seeks are almost always faster than a scan, except on a very small table.
Indexes really only work well when they are *in order*. An index works best when the values in it are listed in perfect sequence, on sequential data pages, as shown in figure 10.1.

The problem with keeping an index in that perfect condition is that indexes, like the data they point to, change. Let’s go back to the traditional paper phone book. What happens when someone new moves into town? Most phone book pages are completely full, so there’s no room to write in a new arrival. In some cities, the phone company periodically issues a supplement, which contains new information: if the original directory doesn’t have the person you’re after, then you flip through the latest supplement to see if they’re listed there instead. As you can imagine, this slows things down, and it’s pretty much what happens when a SQL Server index becomes fragmented.

Imagine that we need to insert the name “Colby” into the index shown in figure 10.1. The value belongs on page 1 of the index, but page 1 is full. So SQL Server *splits* page 1, creating a new page 4. Half the information from page 1 is copied to page 4, and the new value is inserted. The result is a half-empty page 1, and a half-empty page 4. Also, the pages are no longer in *physical sequence*. If SQL Server needs to read the data in logical order, as shown in figure 10.2, it has to read page 1, then page 4, and then skip back to read pages 2 and 3. Over time, this back-and-forth skipping caused by the fragmented index can slow SQL Server down considerably.

In fact, SQL Server internally keeps track of how fragmented an index is, and at a certain point—which depends on a fairly large number of factors—SQL Server will figure out that it’s faster to scan the table for desired values than to use the index. At that point, performance is totally trashed. You’re still wasting time *updating* the index, meaning you get all the downsides of having the index in place, but
you’re not using the index to speed up anything, so you get none of the index’s intended benefits!

**NOTE** If you have a specific query, and want to see if SQL Server is using indexes to execute that query, you can manually run the query in SQL Server Management Studio and look at the *execution plan* SQL Server came up with. I’ll show you how to do that in chapter 12.

Fragmentation occurs to *all* indexes on tables where data is being added, deleted, or changed; the only time fragmentation isn’t an issue is for a table that is entirely read-only. Not many databases have purely read-only tables, so index fragmentation is one of the biggest things that you, as a database maintenance person, need to worry about.

### 10.2 Analyzing index fragmentation

You can quickly check the fragmentation level of an index by opening the index’s properties dialog. However, don’t do it that way.

Checking fragmentation levels in the GUI is slow, and you’re likely going to have to check dozens, if not hundreds, of indexes. This is where you definitely want to shift from the GUI to a query window, because SQL Server can deliver fragmentation information on multiple indexes all at once. Open the query window, and run the following:

```sql
SELECT * FROM sys.dm_db_index_physical_stats(DB_ID('database_name'),
  OBJECT_ID(N'schema.table_name'),
  NULL,NULL,'Sampled')
```

You’ll need to fill in the appropriate `database_name`, such as AdventureWorks2012. By specifying Sampled, you’ll get a good tradeoff between speed of execution and accuracy; specify Limited for a faster but possibly less accurate scan; or specify Detailed for a slower, more accurate look at fragmentation.

You also need to replace `schema.table_name` with the appropriate table name, such as `Person.Address` (from the AdventureWorks2012 database). For example:

```sql
SELECT * FROM sys.dm_db_index_physical_stats(DB_ID('AdventureWorks2012'),
  OBJECT_ID(N'Person.Address'),
  NULL,NULL,'Sampled')
```

You’ll find a column named `avg_fragmentation_in_percent` that shows fragmentation percentage. Note that index names aren’t shown in this output; instead, you get each index’s unique numeric index ID. You’ll also want to look at the `page_count` column, which shows how many pages the index occupies. For extremely small indexes (under a dozen pages or so), fragmentation percentage will never be zero, and will not be entirely accurate. For indexes that small, frankly, fragmentation isn’t much of a concern.
TRY IT NOW  Try analyzing the fragmentation in several tables in one of your databases. Keep in mind that, for especially small tables (which is almost all of them in the sample AdventureWorks database), the fragmentation percentage won’t be accurate, and achieving 0% fragmentation may not be possible.

There are two kinds of fragmentation to worry about:

- **Internal fragmentation**—What I’ve described in this section. It’s when the index’s rows of information are out of order within the database.
- **External fragmentation**—A bit more complex. SQL Server groups 8 KB pages into sets of 8, called extents. When SQL Server needs to read a number of sequential pages, it’s easier (and faster) to do so if they’re all grouped into sequential extents. External fragmentation is when the pages of an index are spread across extents that are separated from each other by extents containing unrelated pages.

### 10.3 Indexes maintenance strategies

Once you’ve determined that an index is overly fragmented (as a rule, anything more than 30% is cause for concern) you need to defragment the index. You have several options, which become more and more flexible with newer versions of SQL Server.

- **Drop and rebuild the index.** This solves the problem, but introduces several of its own. For one, queries that depended upon the index will have poor performance until the index is online. In addition, building an index on a large table may lock the table, preventing changes from users. Here’s the biggest downside if you do this with a clustered index: any nonclustered indexes will be automatically rebuilt on the resulting heap, and when you build the new index all the nonclustered indexes will be rebuilt again. Definitely don’t do this during production hours!

- **Rebuild the index.** This is spiritually the same as dropping and recreating the index, but is done as a single operation in SQL Server. With a clustered index, you won’t cause your nonclustered indexes to be rebuilt on a heap, and then re-rebuilt on the new clustered index. This can still be impactful in production: the index won’t be available for use while it’s being rebuilt, and queries may fail due to locking. There are two ways to perform this kind of rebuild:
  
  - **CREATE INDEX with DROP_EXISTING** uses the CREATE INDEX statement and its DROP_EXISTING option. Indexes that contain constraints (such as a UNIQUE constraint) can be rebuilt using this technique, although under the hood SQL Server has to drop and recreate the constraints, which can be time-consuming. A downside is that indexes have to be rebuilt one at a time, although as you’ll see later in this chapter that isn’t always a bad thing.
  
  - **ALTER INDEX REBUILD** uses the ALTER INDEX statement. Unlike the above option, this doesn’t have to drop and re-create constraints in the index, so it can be faster to run. This is how I usually rebuild indexes.
• **Rebuild the index online.** An option in newer versions of SQL Server (2012 and later), this builds a new index behind the scenes, then uses some sleight-of-hand to swap in the new index and delete the old one. This has a performance impact, but it keeps everything functional and accessible. The big price is in disk space: you’ll need enough to store that second copy of the index while it’s being built.

• **Reorganize the index.** This can be accomplished while the index is still online and being used, because all it really does is rearrange the pages into the correct physical sequence. This process can be paused and restarted, although it may not always help you achieve 0% fragmentation. There’s no impact on nonclustered indexes.

We’ll cover how to rebuild and reorganize an index in a minute. But first, a couple of final notes before moving on.

### 10.3.1 Fill factor

There’s an important concept called *fill factor* that comes into play when you rebuild (but not reorganize) a nonclustered index. The fill factor determines how full (or, if you’re a pessimist, how empty) index leaf pages are in the rebuilt index. A fill factor of 50 means the page is half full (or half empty); a fill factor of 70 means the page is 70% full (30% empty).

Why leave empty space? Empty space allows for the addition of new rows to the index leaf pages without splitting pages, meaning the index lasts longer before it becomes too fragmented to be useful and needs to be rebuilt. But there’s a downside to a low fill factor: while it helps avoid page splits for a period of time, it also slows down SQL Server. Keep in mind that SQL Server reads the entire index page from disk; if the page is half empty, then SQL Server just wasted time reading nothing! For best disk performance, always SQL Server’s weakest spot, you want your index pages 100% full, so that each disk operation is completely useful. Full pages will result in page splits and fragmentation almost immediately, so coming up with the right fill factor is a balancing act. More on that in a moment.

### 10.3.2 Databases that aren’t yours

How much of this maintenance stuff can you do for databases you don’t “own,” such as a database being used to support some line-of-business application?

It depends. You can definitely monitor fragmentation (keeping an eye on things never hurts). Some vendors build index maintenance routines into the application, and so you should see fragmentation go up and down as those routines run. On the other hand, many vendors don’t do anything to maintain their database’s indexes, so you’ll see fragmentation go up . . . and up . . . and up . . . For those, you’ll need to run index rebuilds and reorganizations on your own, probably on a regular basis. Doing so won’t “hurt” the application. In fact, it’ll help it perform much better.
10.4 Performing the maintenance

Although you can accomplish these maintenance tasks using the SQL Server Management Studio GUI, I much prefer to use T-SQL statements, since I can pop them into a SQL Server Agent job and have them run automatically. (You’ll learn more about Agent in an upcoming chapter. Agent is somewhat like Windows Task Scheduler, only for SQL Server).

**NOTE** Because this maintenance is something you should perform regularly, try to get into the habit of using T-SQL to do it. Doing this in the GUI is going to be a bit tedious, and it’s going to discourage you from doing this as often, and as well, as you should.

Note that I’m not offering command examples to drop and recreate an index. I don’t ever recommend doing that as part of routine maintenance.

10.4.1 Recreating an index with DROP_EXISTING

As a quick reminder of what this will do:

- The index will be rebuilt.
- If it’s a clustered index, there will be no impact on nonclustered indexes.
- Index constraints may be dropped and recreated automatically.
- You have to do one index at a time.
- For nonclustered indexes, you can specify a fill factor.
- SQL Server statistics on the index will be updated, so that SQL Server knows the index’s new condition.

With that in mind, here’s the command:

```sql
CREATE [UNIQUE] [CLUSTERED] INDEX index_name
ON schema.table_name(column_name)
WITH (DROP_EXISTING = ON);
```

There’s a lot of information to fill in, so let me offer a concrete example. Suppose you want to rebuild an index named IDX1, on the Person.Address table, and it indexes on the table’s AddrID column. It’s a nonclustered index, but has a UNIQUE constraint, meaning every value in the column must be unique. Here’s what you’d run:

```sql
CREATE UNIQUE INDEX IDX1
ON Person.Address(AddrID)
WITH (DROP_EXISTING = ON);
```

I never use this syntax. I always use ALTER INDEX instead, as it’s easier and can do multiple indexes at once. It also offers the online-rebuild option, if I need it. But you may run into existing maintenance scripts that use this syntax, so you should be able to recognize it when you see it.
10.4.2 Rebuilding an index by using ALTER INDEX

As a reminder:

- The index will be rebuilt.
- If it’s a clustered index, there will be no impact on nonclustered indexes.
- There will be no effect on index constraints.
- You can do multiple indexes with one command.
- You can specify a fill factor for nonclustered indexes.
- SQL Server statistics on the index will be updated, so that SQL Server knows the index’s new condition.

Here’s how to do it:

```sql
ALTER INDEX index_name ON schema.table_name REBUILD;
```

Provide the correct index name and schema.table name (for example, Person.Address) in the appropriate places. Or, to do all indexes on a table:

```sql
ALTER INDEX ALL ON schema.table_name REBUILD;
```

10.4.3 Rebuild the index online

In SQL Server 2012 Enterprise Edition (and later versions), you can also execute this as an online operation, which I described previously. Here’s how:

```sql
ALTER INDEX index_name ON schema.table_name REBUILD
WITH (ONLINE = ON);
```

And, you can specify a new fill factor. If you don’t, the last-used fill factor will be used for the rebuild. Here’s how to specify a new one:

```sql
ALTER INDEX index_name ON schema.table_name REBUILD
WITH (FILLFACTOR = number);
```

Provide a number such as 70 for 70% and you’re good to go. This can be used in combination with an online rebuild:

```sql
WITH (ONLINE = ON, FILLFACTOR = number)
```

10.4.4 Reorganizing an index

As a reminder of what this will do:

- The index will be reorganized, but not completely defragmented.
- If it’s a clustered index, there will be no impact on nonclustered indexes.
- Index constraints are not affected.
- You can do multiple indexes in one command.
- You cannot modify the fill factor.
- SQL Server statistics on the index will not be updated, so you typically want to run an UPDATE STATISTICS command after reorganizing.
Here’s the command:

```
ALTER INDEX index_name ON schema.table_name REORGANIZE;
```

Provide the correct index name and schema.table name (e.g., Person.Address) in the appropriate places. Or, to do all indexes on a table:

```
ALTER INDEX ALL ON schema.table_name REORGANIZE;
```

### 10.5 Suggestions for index maintenance

I’m going to offer suggestions for coming up with the right index maintenance strategy, including the right fill factor for your nonclustered indexes. I have a few assumptions I make going into this, which are based entirely on my own experience with customers. These aren’t hard-and-fast rules, but they’re the guidelines I tend to follow:

- I don’t like to rebuild indexes during production hours, so I try to identify a maintenance window where a rebuild can be conducted.
- When I’m on a version of SQL Server that supports it, I prefer to use online index rebuilds, even though that requires more disk space. Disk space is cheap these days, and online rebuilds avoid the chance that I’ll disrupt some other user or process that’s accessing the database, even during a declared maintenance window.
- I try to minimize rebuilds, just because they do create impact in the database.
- I come up with a maintenance strategy on a per-index basis. That’s because each index on each table tends, in my experience, to be used somewhat differently. If I’m being a bit lazy, I’ll use a one-size-fits-all strategy and choose a strategy that’s good (if not great) for all the indexes, but I try not to be lazy.
- I try to find a good balance between index performance and disk performance.

With that in mind, here’s what I do:

1. I start by rebuilding all of my indexes. For my nonclustered indexes, I choose a somewhat arbitrary fill factor. I grab a number out of the air, usually 70.
2. I monitor fragmentation daily. When I start approaching 30%, I execute a reorganization on the index. If I get to that point quickly—say, in the middle of the workday—I lower my target fill factor.
3. By the time I hit my next maintenance period for an index rebuild, I want to have fragmentation at about 20-30%. If I get to that period and fragmentation is higher than that, I lower my target fill factor; if fragmentation is lower, then I raise my target fill factor.

This often goes on for a few weeks, since in many cases I can only do rebuilds on the weekends. In other words, I want to hit Friday afternoon with 20-30% fragmentation, which means I’m filling up my index pages as much as possible without over-fragmenting. If I hit that 30% threshold midweek, I’ll use a reorganization to buy some time. In scenarios where I have a short nightly maintenance window, I can often
use a nightly index reorganization to keep fragmentation fairly low, but I always try to get that weekly rebuild (at least for busy indexes) to rebalance the fill factor for the upcoming week.

Again, not every index is the same because not every table gets used the same. For example, some databases have lookup tables that contain fairly static data. Because the tables don’t change much, their indexes don’t change much, and I might only run an index rebuild once a month, once a quarter, or something like that. Sometimes I just monitor the indexes and manually run a rebuild when it’s needed, for that kind of table.

The point is to watch what’s happening with fragmentation, and to keep in mind the balancing act between index fragmentation and disk performance.

Go automatic
SQL Server fan Michelle Ufford wrote a complete stored procedure that helps automate some of the index maintenance balancing act. You can find it at http://sqlfool.com/wp-content/uploads/2011/06/dba_indexDefrag_sp_v41.txt.

I want to call special attention to the 30% number that I’ve used a couple of times. That’s just a kind of general rule that the SQL Server community uses; it does not mean that an index with less than 30% fragmentation is just fine. Any fragmentation slows things down; with an especially large index, 30% fragmentation may well be too high for the index to be useful. We ideally want 0% fragmentation, but we can’t practically achieve that, so we learn to live with a little fragmentation, and we actively maintain the indexes to minimize how much fragmentation we have to live with. Nothing but experience will tell you what’s acceptable for a given application in a given environment.

10.6 Hands-on lab
For this chapter’s hands-on lab, try to perform an index reorganization and an index rebuild. The commands were provided in this chapter; if you’re using the AdventureWorks sample database, then odds are you won’t find much fragmentation or see your rebuild or reorganization have much effect. That’s because AdventureWorks doesn’t contain much data. But becoming comfortable with the maintenance commands is a good reason to try ’em out, anyway.

Because the command syntax is provided in this chapter, you won’t find sample answers for this lab on MoreLunches.com.
Learn SQL SERVER ADMINISTRATION IN A MONTH OF LUNCHES
DON JONES

Microsoft SQL Server is used by millions of businesses, ranging in size from Fortune 500s to small shops worldwide. Whether you’re just getting started as a DBA, supporting a SQL Server-driven application, or you’ve been drafted by your office as the SQL Server admin, you do not need a thousand-page book to get up and running.

Learn SQL Server Administration in a Month of Lunches is the perfect way to get started with SQL Server. This concise, easy-to-read book skips academic introductions and teaches you day-to-day techniques for maintenance, backup and recovery, performance monitoring, and more. Each of the 21 short lessons gives you practical takeaways you’ll use over and over.

What’s Inside
- Master the basics—indexes, logins, backup, recovery...
  and more
- Learn what you can and cannot do when supporting
  a third-party application
- Monitor and improve performance
- Written by expert trainer and bestselling author Don Jones

Accessible to readers of any level of experience, the book covers techniques for all versions of SQL Server 2005-2014.

Don Jones is a Microsoft MVP, speaker, and trainer. He is the creator of the Month of Lunches series and author of over 50 books on PowerShell, IIS, Active Directory, SCCM, SQL Server, and more.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/LearnSQLServerAdministrationinaMonthofLunches

MANNING $34.99 / Can $36.99 [INCLUDING eBook]