# Course Overview

## Course Overview

Hi, everyone. My name is Gail Shaw. Welcome to my course, Optimizing SQL Server Statistics and Indexes. I'm a technical lead at Intellect in Johannesburg, South Africa. In this course, we're going to have a look of why and how you should perform index and statistics maintenance in SQL Server. Some of the things we will cover include creating and dropping indexes to keep up with the changing workload, rowstore index fragmentation and how to handle it, why columnstore indexes need maintenance and what options you have, and what statistics are and why they need maintenance. By the end of this course, you'll be well versed on what you need to do to keep your indexes in order and your database performing well. Before starting the course, you should be familiar with SQL Server indexes, what they are and how they're used. I hope you'll join me on this journey to learn more about indexes with the Optimizing SQL Server Statistics and Indexes course at Pluralsight.

# Overview of Index Maintenance

## Course Introduction

Hello, and welcome to this course on Optimizing SQL Server Statistics and Indexes for Pluralsight. Index maintenance is something that's very important for the health of your database. But in the years of looking at production systems, I generally see it either underdone or not at all or badly overdone. And in this course, I'm going to address both of those and hopefully give you some ideas of what is needed in terms of maintenance and what's not. But first, let me look at what we mean by maintenance at all. Maintenance in this sense, index maintenance, is a form of cleanup. Relational databases are typically optimized for reads. We do the writes quickly, knowing that we can clean it up later because we don't want the writes to impact our reads. Maintenance is something that's often done, or should be done I should say, outside of peak business hours. Doing this kind of maintenance while users are actively using the system is definitely going to degrade their experience, and so this is usually something that's done during maintenance windows, weekends, late at night, that kind of thing. And the other thing is that the kind of maintenance you need and the frequency that you need it at is very much determined by the workload. A system that's read only is almost certainly not going to need index maintenance. Or if it does, it's going to be very little and very specific maintenance. A workload that's mostly writes, however, is likely going to need a lot of maintenance. So what you need and what you should be doing is very much dependent on the type of workload you're running on the database system. Index maintenance is just one aspect of good database maintenance. I'm not going to talk about the other two beyond the slide. But for you awareness, there are a couple of other things that you need to be doing to your databases. The first thing you need to be doing is backups, lots of backups. And make sure you can restore them because a backup you can't restore is a complete waste of space and time and energy. Test your backups. Make sure you can restore your backups, and, absolutely, you must have backups. The second thing you need to do is consistency checks. This is to make sure that the I/O subsystem hasn't scrambled part of your database at some point. Consistency checks are also something that should be done relatively often. How often is often determined by data loss allowance and/or backup strategy. I'm not going to get any more into that right now. There are other courses of Pluralsight on consistency checks. For the rest of this course, what we're going to be focusing on is looking at missing and unused indexes, how to detect them, test them, and what you should do about those. We're going to look at rowstore indexes and have a look at the type of maintenance that's necessary for those indexes. We're going to look at columnstores, see what maintenance those needs because it's different from most or indexes and how they should be maintained. And finally, we're going to look at statistics objects, see what they do in the first place, why they're important, and what kind of maintenance we should do on them. I hope you'll join me for this

journey through the types and methods for SQL Server index and statistics maintenance, and I look forward to seeing you in the next module.

# Review Index Usage and Identify Potential Missing Indexes

## Why Do You Have Missing Indexes?

Hello, and welcome to this module on reviewing index usage. This is the first aspect of what I consider to be good index maintenance, and it's the one I find that's most often ignored in production systems. We'll start by looking at why this kind of index maintenance is even needed. And after that, I'll look at the three aspects of reviewing index usage, missing indexes, unused indexes, and redundant indexes. So why do we need to do this? Surely if we do the index design correctly when the system has been developed, then the indexes will be correct for the system. I mean, it's not as if they decay over time or anything. Well, the indexes don't decay or morph, but the system often does not. Not decay, we hope, but change. The workload run against the database changes over time, gaining new queries, changing the frequency that the existing queries run, and sometimes removing what were once important queries. The data in the database changes over time. The volume and distribution of data will change. Data may get archived. And if the system uses soft deletes, then the ratio of normal data to deleted data will also change. And finally, the schema itself may change. Applications morph over their lifetime. Features get added. Features get removed. I've almost never seen tables or columns actually removed from a database. But new ones added to the database, that happens very often. The three aspects of reviewing index usage that I'm going to discuss are missing indexes where there are queries that could benefit from indexes that don't exist, unused indexes where there are indexes that no queries use, and redundant indexes where two or more indexes could be consolidated in to one with no decrease in performance. The first two of those will come about naturally as systems change and data changes. Ideal data access paths change, and the queries run against the database change. The third, however, will only be necessary if the indexes have been created without sufficient analysis at some time in the past, resulting in overlapping indexes. This is quite common to find when you inherit existing systems, whether they be vendor systems or custom developed ones. Let's start with identifying missing indexes. There are two main tools for identifying missing indexes, the missing index Dynamic Management Views, usually abbreviated as DMVs, and the Database Tuning Advisor, often called DTA. I need to emphasize that neither of these are perfect. They both produce index recommendations, but they are recommendations, and they should be tested carefully before implementing, lest you end up with huge amounts of unused and redundant indexes. The worst case of overindexing I have ever seen was a result of people running the Database Tuning Advisor against their test environment and then implementing the recommended indexes in production with no testing, at least three separate times. The result was a transactional table that had to

get very fast inserts, having 75 overlapping and unnecessary indexes. The only way to fix that was to drop the whole lot and reindex that table from scratch. We'll start by looking at the missing index DMVs. The clever part here isn't the DMV itself. That's just one of the ways the index recommendations are exposed. The recommendations are generated by the query optimizer as part of optimizing queries. The disadvantage here is that relatively few things are considered compared to what the Database Tuning Advisor will do. For example, the missing index DMVs will not recommend partitioning, clustered indexes, or columnstores; neither will they recommend unique constraints or unique indexes. The origin of the missing index recommendations is, as I mentioned, the query optimizer. The way the process works is that a query or batch is submitted to the optimizer. A query plan is then generated. This gets executed. The optimizer then logs information on what index it could have used for that query had it existed. It will only recommend one index per query at a time. These recommendations are then stored and made accessible both in the query plan that was generated and from the missing index database.

## The Missing Index DMVs

If we look at the missing index DMVs, they are unfortunately a tad on the complex side. I'll refer you to the official Microsoft documentation for the full details on exactly what is in the three DMVs, but this is a query to get the details of the missing indexes. It's taken mostly from the Microsoft documentation with some minor changes. What this will produce is one row for each recommended index. The first column is what database that image should belong to. The second is the schema of the table, DBO if there's no user schemas. And then the third is the table name. The other columns take a bit more explaining. The avg_user_impact is a measure of the estimated improvement that the index will give. Key word here is estimated. These recommendations are generated by the query optimizer. And hence, any costs that it uses to calculate improvements are estimates, and so any measure of improvement is also going to be an estimate. There are columns in the missing index group stats DMV that are updated after the queries execute and are hence based on executed queries, not just compiles. But the costs themselves are still estimates. The equality_columns column is a comma-delimited list of column names that should go into the index. These are columns that the query that triggered this recommendation had as part of equality predicates in its WHERE clause or joins. The inequality_columns list is similarly a comma-delimited list of columns that the query triggering this recommendation had as part of its inequality predicates. If you've watched my course on index design, you'll know that columns used for equality predicates should go before columns used for inequality predicates in the index key, and that's why these columns are referenced separately. The include_columns is also a comma-delimited list of columns that the triggering query had in the SELECT clause, GROUP BY, or other places within the query that can't be used as predicates. This is hence appropriate for putting in the indexes include_columns. One large caveat here, the recommendations are, in most cases, fully covering indexes, eliminating the need for key lookups. If your system has a lot of queries that use SELECT * or has Entity Framework queries that materialize the entire object, then your missing index recommendations will often have all of the columns in the table in the index recommendation spread

between the three column specifications. This might not be ideal, not unless you like your database to be 10 or 20 times the size of the actual data in it. As I've alluded to, there are some limitations to these index recommendations. The first is that the optimizer doesn't consider expanding existing indexes, nor does it look through the list of currently recommended indexes when adding a recommendation. Hence, it's entirely possible for a table to have an existing index on column 1 and for there to be 2 index recommendations for indexes on column 1 include column 2 and column 1 and column 2 both in the key. In this case, widening the existing index by adding column 2 to the key would be perfectly adequate; whereas, taking the recommendations and implementing them without analysis would result in redundant indexes. The second limitation is that the evaluation is based on a single query execution, not your workload. You might have a query that's really expensive and runs once a month. And if it runs and they are missing indexes, they'll get recommended, even though overall that query is probably not that important. There's a limited number of indexes kept in the DMV. No more than 1 is recommended per query at a time, and there's a limit to 600 rows in the DMV. That's 600 spread across all of the databases on your instance. These DMVs are also memory-only and so are lost when SQL Server restarts. Finally, there are things that the query optimizer simply doesn't recommend. It won't recommend clustered indexes. It won't recommend columnstores, clustered or non-clustered. It won't consider partitioning. It won't consider unique constraints or unique indexes. This is the straightforward, simple recommendation of non-clustered indexes, nothing more.

## Demo: Identifying Missing Indexes via the DMVs

With that covered, let's take a look at index recommendations from the missing index DMVs, see how they're generated, evaluate the results of those missing indexes, test the indexes, and see their impact on the query execution. I'm going to start by running a workload against my demo database. Now this is just a test workload, and it can't really simulate real users. It's very predictable. And you'll notice in the demos that we're going to see some very strange numbers in a lot of these DMVs. That's a side effect of this being a demo database and a simulated workload. And once that's been running for a while, I'm going to have a look at the missing index DMVs. The three DMVs that you're interested in here are sys.dm_db_missing_index_groups, sys.dm_db_missing_index_group_stats, and sys.dm_db_missing_index_details. We run that. And at this point in time, there are three missing indexes on this database. Well, there are three missing indexes in the DMVs. There are not three missing indexes. The query optimizer will only recommend one index per query. And so if we've got to query that could use multiple indexes, we're going to see the most important one here, not the others. I have two missing indexes on my Shipments table and one missing index on my ShipmentDetails table. Something to note immediately is that the equality_column is the same for the two missing indexes on Shipments. This is one of the downsides I mentioned earlier. The optimizer does not consider existing indexes or existing recommendations when it's suggesting indexes. So what we've got here are two recommendations for something that could actually be a single index. That was the one way to see missing indexes in the database. The second way is to look at the execution plan of the query. Generally, which one

you do is going to depend on how you got to this point in tuning. Did you start by looking for missing indexes, or did you start by trying to tune a query? If you started by looking for missing indexes, you're going to go through the missing index DMV. If you started by looking at poorly performing queries, then you're going to come across the recommendations in the query plans first. I do prefer the query plan because then I can see which query is going to benefit from that index. If you recall looking at the missing index DMVs, there was no indication which queries were missing indexes. Coming from the query plan, you're tuning a query, so you can see it's missing index. And I'm going to go and create that index now. Please, Please, please, please do not keep the existing name. Yes, I have actually run across production databases with indexes named Name of Missing Index, sysname. That is the naming convention that I like for indexes, table name followed by the key columns. But please, if you have a name extender for indexes already, use yours, not mine. With that index created, let's go back and have a look at the missing index DMVs, and you'll notice that my missing index on ShipmentDetails is gone. I've now got a missing index on transactions instead. This is from the same query. The first recommendation was for a missing index on ShipmentDetails. Once that was fixed, the second recommendation is for one on Transactions. Now it just remains to see how much of an improvement that index had. This time I'm going to use Query Store for this. In a later demo, I'll do the analysis using statistics I/O and time. But for now, I want to show you Query Store. The view that is going to get me that query is the 25 highest resource-consuming queries in my database. If the query you're looking at is not one of the top queries, then you'll probably need the tracked queries view instead. If I find that query and refresh Query Store, I can see it's got a new plan, a plan which is much lower in duration, much more efficient because it's had that index created. And if I scroll across in the query plan itself, you can see where that new index has now been used.

## Database Tuning Advisor

The second way to get index recommendations is via the Database Tuning Advisor. This tool has been around a long time, since at least SQL Server 2000 if not before. It's evolved over the years, and it's a lot better than it used to be. The main difference between this and the missing index recommendations is that this is a manual process. It's not something that happens automatically in the background as your system is in use. It can produce recommendations for a single query or for a defined workload. I strongly recommend if you're going to use DTA, use it to tune your workload. If you recall in the previous section, I mentioned that one of the downsides of the missing index recommendations is that the optimizer considers a single query at a time. DTA is capable of tuning an entire workload, and that will in general produce better recommendations. Hence, it should be run that way wherever possible. One thing to be aware of is that DTA will test out the index recommendations as part of the analysis process, and it will do this on the target database. If you run DTA in the middle of the day and point it at the production database, there will likely be some consequences, impact to your user workload to start with. The last important thing to note is that DTA is very prone to over recommending indexes and statistics. I've before now tuned

a single query and got five index suggestions and multiple statistics suggestions. Please, please, please do not automatically create the recommended objects. Test them out. See which give the best results. If, for example, four indexes are recommended, and creating one gives us 70% improvement in query performance and then the other three combined give you another 5%, creating just the first one is likely to be your best approach. You do not want to overindex your database. The workflow for DTA consists of three phases. These can, in most cases, be done on separate instances of SQL Server. The first is to generate the workload. This can be done in a couple of ways, and I'll cover those in a moment. This should be done on your production server. That's the server you're trying to tune. There's no point in tuning a development or test workload. The second is your analysis. This should not be done on the production server as it adds overhead, and DTA will create and drop indexes in the process of doing its analysis. The creation of indexes may cause blocking, and creating and dropping objects may violate your change control policies. Do this on a test server, but make sure that the database on the test server is a copy of production in every way, especially data volume. What I recommend here is if you've taken a workload from production, take a backup around the same time. That way your workload will actually work and won't be referencing missing rows. The third phase is the implementation of the recommendations, and I strongly recommend not doing this automatically. Test out the recommendations, decide which will be implemented, and then implement them using your normal change control or continual integration process. Having indexes in production which are not on your dev or test servers can cause all sorts of pain, including the deployment of duplicate indexes when people do more index analysis on dev. Recent versions of DTA have four options for generating a workload. The first is Query Store. This, of course, requires that Query Store is turned on on the target database and that it has been on long enough to capture a representative workload for the database. Please don't turn Query Store on and then run DTA a couple of hours later. You need time to capture a representative workload. Since Query Store is stored in the user database and included in the backups, you can take a backup of the production database, restore it to your test environment, and then run DTA with the Query Store option against that restored database. The second option is a manual T-SQL script consisting of the queries that need to be tuned. This is probably not a good option. It's extra work to generate the script. And other than allowing the workload to be generated on a different server to where DTA is run, it doesn't offer a lot of benefits over the other options. The third is plan cache. In this one, DTA analyzes the query in the plan cache of the target instance and uses the execution statistics associated with those to come up with a workload. I'm not overly fond of this option. There are too many things that can cause queries to be evicted from cache or never cached at all. And this one has to be run on the production server because you can't move a plan cache to a dev environment. The last option is a profiler workload. This is unfortunately still profiler, not extended events. Perhaps one of these decades, we'll get rid of profiler entirely, and this will use extended events. Maybe. This along with Query Store is my preferred option. I urge towards using this for SQL instances of 2014 or lower or where Query Store is not. And use Query Store for SQL instances of 2016 and above where Query Store has been turned on long enough to have a good collection of the workload. DTA is not perfect. It has limitations and downsides like everything else. The first is the need for a comprehensive workload. Because it doesn't run automatically in the background like

the missing index recommendations do, it's dependent on the captured workload being complete. If the workload presented a DTA is incomplete from Query Store being turned on recently or flushing old queries, from the captured trace being a subset of what actually runs, from a non-representative plan cache, or from a workload script that's incomplete, then the indexes recommended will not be ideal for the workload. and with some of the options for DTA, that could be very problematic if implemented without testing. DTA adds load to the target server because it will be creating the indexes and testing the workload. Please do not run it against a production server ever. The last one I've alluded to already. DTA tends to over-recommend quite badly. Rather, don't have it automatically implement the recommendations and then test them out yourself before deploying the changes.

## Demo: Database Tuning Advisor

So in this demo, I'm going to run through all the options for DTAs workload generation. I'll then show you how to run an analysis session, look at the various options, and test out the recommendations. To show off DTA, I need a workload running, and I'm going to use the same simulated workload that I used in the previous demo. Database Tuning Advisor can take its workload from a number of places. In this demo, I'm going to show you Query Store and trace, which are my two preferred options. The Query Store is going to be capturing the data anyway because it's automatically capturing everything running against the database. The trace though I need to do manually. So, SQL Profiler it is. The template that you want to use here, or should I say need to use here, is the tuning template. Don't make any modifications to it. What's in that template is what DTA needs. Just select the template and run the trace. Or preferably, if this is a production system, and I assume it's a production system, script out the trace and run it as a server-side trace rather than using the profiler GUI. It's incredibly easy to crash a production server using the profiler GUI. Whether you're using the GUI or server-side trace, the result will be a .trc file. I'm going to copy that along with the backup of my database to my test server, restore the database there, and then we can run DTA from a test server. This is my test server, or more correctly in my case, this is my VM in Azure because I don't exactly have a test server. What I've done is restored my database and then changed Query Store setting to read only. I don't want to risk any of the tuning workload getting picked up by Query Store as if it was a real workload. The Query Store already contains everything that ran against my production database because it's in the database, it's in the backup, and this is a restored copy of my production database. But I don't want it to be modified any further, hence read only. With that then, we can start DTA. Connect it to your test server please, not your production server. The first choice is where's the workload coming from? Let's choose Query Store. And it wants to know which database is Query Store to use. You could restore a copy of the database from production to use as Query Store and then restore another copy to run the analysis on if you want. It strikes me a little odd if you're running this on a test server, which you should be. And you'll note that you don't have the option to use databases on different servers, just two databases on the same instance. In this case, this is my test server, and so I can use the same database for the analysis and the tuning. Let's look at the tuning options. There's a lot of options here. The defaults are mostly good enough for most

cases. If you look at the first section, this is our physical design structures. This is what you want DTA to recommend. The default is indexes, which is all indexes clustered and nonclustered. You can set it to also recommend indexed views. I use indexed views so seldom, I've never seen a use for that to be honest. When I do want indexed views, I tend to test them very specifically myself. But if you're not sure, that might be a good option. You can set it to include filtered indexes. I didn't do that in this demo. But if you've got a system that does soft deletes where a deleted row is indicated by an is deleted column or deleted date column, then selecting Include filtered indexes will often be extremely useful because DTA can pick up the fact that most or all of your queries are filtering for this deleted column and hence create filtered indexes, which is a fantastic way of dealing with systems that have soft deletes. Indexes, as I said, is the default. That's clustered and nonclustered. If you only want it to suggest nonclustered indexes, you can select that is an option as well. You can ask it to recommend column stores. That's also not a default. I recommend doing this. Column stores are extremely useful these days, and they don't have the downsides they had a few versions go of being read only. And besides, you're going to test all these recommendations out anyway, so you may as well see what it's going to suggest. The last option is to evaluate only the existing indexes. What that will do is generate recommendations for indexes to be dropped, not created. I really don't like using DTA for that. My workload has got to be comprehensive if that's going to give me good recommendations. I don't trust that my workload is comprehensive. I probably did not run it for a month. I hope you didn't run it for a month. I'd hate to see the size of that file. There's other ways to pick up unused indexes. I recommend not using DTA for that. Second section, partitioning. For the most part, I'm going to say rather say no partitioning because partitioning is not something that you do for performance reasons. Partitioning is something that you do for data management, if you need to load data quickly, remove data quickly, do index maintenance on partitions, do compression differently on different sets of your data. That's not something DTA can pick up. The last option is what do we do with existing indexes? Do we consider dropping them? Do we keep them? Do we keep certain of them? Rather, keep them all. I don't like using DTA to recommend removing indexers. The advanced tuning options doesn't give us very much. The main thing that you might like to change is whether the recommendations are offline or online. If you've got no downtime to make the changes, you might want to say give me the recommendations online so there will be create index with the online options. That's not a big deal though. You can always change that yourself before running the recommendations that work on production. I'm not going to run the analysis from Query Store. I'm going to run it from the trace that I created earlier. So switch my workload to File and go and find my trace file. Once I've got the file selected and the database ticked, click Start Analysis and go make some coffee. This can take a while. In my case, it took seconds, but that's because I've got a demo database, a tiny workload and virtually no data. Your mileage will vary. On a large database, this will take time. And once it's all finished, we can look at the recommendations it's made. You'll note it's made a lot of recommendations. My workload consists of four queries. I've got more than twice that number of recommendations. This is why I say DTA tends to over-recommend quite badly. We've got six indexes and five stats objects recommended here, and let's have a look at the columns that they're on. One thing you can see here is that we've got a duplicate recommendation. Of the four indexes recommended for the Shipments table,

two of them have OriginStationID as the key column. Same problem that we saw with the missing index DMVs, just dialed up to 15. You probably don't want to create duplicate indexes. There are certainly edge cases where they're useful, but that's edge cases, not normal. So let me generate a script for these, and then we can have a look at them in Management Studio. One thing to note. Once you open this in Management Studio, the recommended index names are horrible. The numbers there are the database ID, 5, the object ID, which for Shipment starts with 141, and then the column ID of the key columns, followed by the column IDs of the include columns. It's just plain horrible. Before you create these on your production system, please change the names to something sensible, something understandable, something that adheres to your naming standard for indexes. For initial testing, I'm just going to select the two indexes on Shipments that have OriginStationID as their key columns. Let's see whether these are both needed or whether I can make due with just the larger index. To figure that out, I need to identify which query this recommendation came from. You'll note DTA didn't tell me that. So, I'm going to have to do a little bit of manual work, again with Query Store. I'm looking for the query that filters on OriginStationID or potentially joins on it. If you've got a large workload, this could be really painful. But fortunately, I've only got four queries, and so it's pretty easy for me to find which query it is. It's a gem of a query, and it's created from Entity Framework. In the previous demo, I used Query Store to test the improvement for the query. But Query Store on this database is read only, and so I'm going to rather use statistics I/O and time to do a comparison. First, the query without the indexes, 32 ms of CPU time, 140 ms elapsed time, and a 1,358 reads on my Shipments table. If I create the two indexes that DTA recommended and look at the statistics again, I've got a vast improvement, 0 ms of CPU time, 117 ms of elapsed time, and my reads on the Shipment table are down to 55. This is really good. The question is do I need both of those indexes? Let me drop that narrow index on just OriginStationID and run the query again. It's slower. It's definitely slower. I have dropped an index that I wanted to use. But it's not significantly slower. I've gone from 117 ms of elapsed time to 139, and I've gone from 55 reads to 76, and I've saved an index. I don't have another redundant structure in my database costing me insert time, update time, and index maintenance time. I would probably take that tradeoff and only create the wider index in this case. This query is not that important. I don't need that 20 ms elapsed time. Note that's not CPU time. That's world clock elapsed time, and most users are not going to notice 20 ms. The increase in reads is not huge. So in this case, I take the additional time and not create the duplicate index. And if this was a real system and a real analysis, I'd repeat that process for the other indexes that were recommended. I don't usually create stats from DTA. You can. I find they're not all that valuable to be honest. But if you were going to test them, you'd test them the same way. Identify which queries could use them, run the query, test it, run the query again. With the indexes created, test again. Repeat until you're happy that you've created all of the useful indexes.

## Summary of Missing Indexes

That's our missing indexes covered. This is something that needs to be done somewhat regularly with a system that's getting enhancements and changes and occasionally with systems that are static in terms of feature changes,

but still in use and hence have changing data. The two options here are the missing index recommendations with the DMVs being the main way to get those recommendations and the Database Tuning Advisor. Neither of them is going to do the full job without supervision. You will need to do some testing, you will need to do some work, and you should ideally decide what's going to be implemented yourself and not leave it up to the tools.

## Identfying Unused Indexes

The next thing on our list of reviewing indexes is identifying unused indexes. The source for these is often the same as missing indexes, data changes, schema changes, and workload changes. It's also possible that unused indexes were created by somebody in the past, and they're sitting in your system, and they're never going to be used because they're just not suitable for your queries. SQL Server does track index usage in a DMV called the index usage stats. This, like most DMVs, is transient. It's memory only. And the data will be lost when the server is restarted. Hence, it's important to do this kind of analysis work when the server has been running for some time. How long? Well, that depends on the business cycle for your system. If your system has clear month-end processes, then you want the server to have not been restarted since before the last month end so that those stats are included. Similar, but much harder to achieve, if you've got a year-end process. As with so many things around indexes, the DMVs provide useful guidelines, not directives, to be implemented without consideration, mostly due to the transient nature of this data. Here's a simple query against the index usage stats DMV. I've done a left join from sys.indexes because an index only shows up in the DMV once it has been used at least once for some reason, read or update, since the server was last started. Hence, unused indexes of static tables won't be present in the DMV at all, and they can easily be missed. The columns of interest here are user_scans, user_seeks, user_lookups, and user_updates. There are system versions of each of these, but I find that those are not all that relevant. I'm interested in how the user queries are using the indexes, not how the system's using them. User_scans is the number of full scans of the index since the last restart of SQL Server. User_seeks is similarly the number of seeks. User_lookups will be 0 for nonclustered indexes. For a clustered index, it's the number of key lookups done to that index. High values for a clustered index don't tell you anything about the clustered index, but it does suggest that one or more of your nonclustered indexes on that table might need to be revisited and possibly have some include columns added. User_updates is the number of modifications made to that index. This will be inserts, updates, and deletes combined. If this is high and seeks and scans are low, then the index is adding overhead for little purpose and is a good candidate to be dropped. Unique indexes should never be dropped. They're enforcing constraints on the data. The fact that they're indexes is a secondary consideration. In general, removing unused indexes is a good thing. It frees up space in the database. It reduces the overhead on data modifications because now the unused indexes don't need to be kept up to date. The downside is that analysis is needed to determine whether or not an indexes really unused. Consider an index that's critical for a year-end process and never used any other time. If it's dropped, then that important query could suddenly run poorly and impact your year-end process. You definitely want to be careful

dropping your indexes. Those changes should be carefully tested. If the application has automated tests, then it may be sufficient to make the changes in the test environment and run the automated tests, watching for query performance degradation. If not, I recommend using something like distributed replay to test and ensure that there were no unacceptable performance degradations.

## Demo: Using the Index Usage DMV

In this demo, I'm going to go through what the index usage DMV shows and evaluate some of the indexes to see whether they're used or not. For doing the unused index analysis, it's important that the database has been running for some time. That's both the instance and the database. So you don't want to have taken the database offline or closed it recently, nor do you want the instance to have been restarted not long ago. This is because the unused index information is memory-only and is lost on a restart of SQL Server or any other time the database is closed. I've got my workload running in the background again, and that's sufficient for this system. I've also run a whole bunch of updates and inserts, although being a simulated workload and a demo system, you'll notice that the numbers are quite round. Still, when I render the index usage stats, I can see interesting information here. I can see that a lot of my indexes have had seeks and new updates, and I can see that I've got some indexes which have had updates, but I've never actually been used. An example here is the index on the Shipments table on DestinationStation, 1,500 updates and no seeks or scans. Another index which doesn't appear to have been used is the index on the Transactions table, idx_Transactions. That's had no updates, no seeks, no scans at all. In terms of severity, the first one's a bigger problem than the second. The first one has been costing us in certain update time, as well as taking up space in the database. The second one has just been taking up space in the database. In both cases though, those indexes are not needed. Or at least they don't appear to be needed based on what we're seeing in these indexed DMVs. Before I drop them though, I want to make sure they haven't been used. For this, I'm going to use Query Store. If you don't have Query Store turned on in your database, then you're going to need a long-running trace or analysis of the plan cache or some similar long-term analysis to make sure that that index has not been used. I have Query Store turned on, and so I'm going to query the Query Store tables and see if that index is in any of the plans that it has stored. This is still not comprehensive. Query Store doesn't go back to the beginning of time, but it's probably good enough, and I'm going to test this anyway. So we're going to search through the Query Store plan table for any plans that have got that index name in them. And I do indeed get three queries that have used that index. So I'll take their query IDs and going query the query_store_query_text_table to see what those are. And they are all inserts. This confirms what I see from the index usage stats DMV. That particular index has been used for modifications only, not for seeks or scans. So it's been updated, but never used usefully. Given this, I would make a good case for dropping this index, though on an important system, I might want to do some additional testing. Alternately, I could drop it and look for consequences. If you don't have the ability to test, that's second best. But it's often a case where you don't have the ability to test. If I drop some indexes and they are needed, what I would

expect to see is an increase in the overall resource consumption reports or an entry in the regressed queries report for queries that actually needed that index.

## Identifying Redundant Indexes

The last section in revising index usage is redundant indexes. Unlike the other two aspects, this isn't something that happens due to workload, schema, or data changes. It comes from creating indexes without sufficient analysis or without a clear understanding of how indexes work. A duplicate index is one where either the key columns are identical to another index and in the same order or where the key columns of one index are a left-based subset of the key columns of another index and in the same order. We might have an index on column 1, column 2 and a second index on just column 1. A query that could use the second index can use the first one just as well. The first index is slightly larger, but it's extremely uncommon for that size difference to make a measurable impact on query performance. Uncommon, not impossible. There are definitely circumstances where you might want a redundant index, but they're rare, and they're definitely worth testing carefully to make sure you really do have one of those situations. Note that I'm not looking at include columns in my definitions of redundant indexes. They're easy enough to handle when consolidating indexes because the order of the include columns doesn't matter, unlike the order of key columns. You can just take the include columns of the two indexes and combine them. This is the script that I usually use to identified duplicate indexes. Now it doesn't show duplicate indexes. It shows all of the indexes in a database with their key columns in a delimited list. I prefer working from a full list to make my decisions about index consolidation rather than trying to write a script that will do it perfectly for me.

## Demo: Redundant Indexes

So in this demo, we're going to have a look at our database and identify some duplicate indexes, look at the queries that use them, and then consolidate those indexes and show that the queries can use consolidated indexes perfectly well. For finding duplicate indexes, I don't need one of the DMVs. Rather, I want the system views themselves. Specifically, I want the system views around the indexes and index columns. That will be sys.indexes and sys.index_columns. This query will produce a list of indexes in the system with a comma-delimited list of their key columns. I'm not looking for include columns, just the key columns for the moment. So this produces 29 rows. That's all the indexes, clustered and nonclustered, that I have in this database. This isn't finding duplicate indexes. This is giving me a list that I can use to find duplicate indexes. And reading down the list, we do indeed have two indexes which are duplicates of each other, both on the Shipments table, both with OriginStationID as their key column. They have no other key columns, so this is the first example of a duplicate index, one where the key columns are identical and in the same order. So let me script those two indexes out so I can have a look at them in detail. One of them is just on the OriginStationID. One of them is on the OriginStationID with a long list of include columns. That doesn't

change the fact that these indexes are duplicates. The key columns overlap. When I consolidate them, consolidating the include columns is easy. It's the key columns which define that indexes are duplicate. Now there's one interesting thing to note here. If I go and look at the index usage stats, both of these indexes have been used. That's not uncommon. Even if these two indexes were complete and total duplicates of each other, key and include columns, they would still both be in use. The optimizer doesn't pick one only to use. It will use both of them. But it doesn't need to use both of them. So let's go to the Query Store quickly and see where these indexes are used, which queries used them. I can see that two queries have those indexes referenced in their execution plans, so I'm going to take them and test them. The first one doesn't actually use those indexes at all. It's a straight table scan on the Shipments table. So, while that index name may have appeared in the execution plan and that execution plan is quite complex, the query doesn't actually use it. So I can ignore that query. I don't need to test it to see if I can drop these indexes. The second one's more complicated, and I can see it is using the index. In fact, it's using both of them. I'm not going to use Query Store to test out performance differences. This is going to be a straightforward comparison of the I/O and time statistics. So first, let me run the query with both of those indexes in existence and get the I/O statistics and the execution time. And then I'm going to remove the redundant index. I'm removing the index ahead no include columns because it is this smaller of the indexes, and it is the one that's less useful. Queries will use those include columns. If both queries had include columns and different ones, what I would do is create one index with a combined list of include columns and then drop both the original ones. And if I run the query again after doing that, I can see the at the I/O stats are indeed slightly up, not significantly. I've gone from 87 logical reads to 122, which, unless this is a really important query, is probably fine. Execution time has stayed much the same. The thing to note with the time statistics is that those numbers are not completely accurate. There is an error on them. If you run the same query, you will often find a 5 or 6-ms difference in both the CPU time and elapsed time. So don't take small differences as really important. They're not. In this case, the CPU and elapsed times are similar enough that I can be fairly confident that dropping that redundant index is not going to hurt my query performance.

## Summary

In this module, we've had a look at what kind of index maintenance is required to keep indexes optimal during the lifetime of a system. The re-evaluations necessary are because workloads change, data changes, and schemas change. The three main aspects to revising index usage are missing indexes where a query could use an index if it existed, but it doesn't, unused indexes where the index could be used by a query, but it isn't because there are no queries that could use that index, and redundant indexes where two or more indexes could perfectly well satisfy a query, and the query is only going to use one of them. And hence, the second one is completely redundant and unnecessary. In the next module, I'll have a look at what's considered a more traditional index maintenance, index rebuilds and index reorganizations. See you there.

# Maintaining Rowstore Indexes

## Introduction and Cause of Index Fragmentation

Hi, and welcome to this module on maintaining rowstore indexes. In the previous module, we looked at evaluating your databases for missing indexes, unused indexes, and redundant indexes, things that aren't usually the first things that come to mind when talking about index maintenance. In this module, I'll be talking about more traditional index maintenance, specifically for rowstore indexes. Columnstores have somewhat different considerations, and so they'll be covered in a separate module. The first thing I want to look at is why we need to maintain indexes in the first place. This isn't going to be the same reason as for the index changes we discussed in the previous module. But again, indexes don't decay by themselves, and so I want to look at what it is that results in the need for maintenance. After that, I'll show you how to identify which indexes need maintenance and discuss some of the great debates around index maintenance. Finally, we'll look at how to fix the problems identified, the various methods and options that you have for index maintenance. Why then do we need to maintain indexes at all? First for indexes, it has to do with the index tree structure, specifically around the ordering of the leaf levels. So let me do a quick recap of index structure. If you want a more detailed discussion, see my course on index design. An index consists of a root page, always a single root page, and then zero or more intermediate pages organized in zero or more levels and, finally, one or more leaf pages, which contain the actual row in the case of clustered indexes or the index key and include columns for nonclustered indexes. The leaf pages are what we're interested in here, so let's focus on those. The rows in the index leaf level are logically ordered by the key. In this case, that's a child 1 column. Here we have a perfectly structured index leaf level. The logical order of the pages, as determined by the index key, matches the physical order of the pages in the file. All the pages are equally full, so there's no wasted space in the index. But what happens when rows need to be inserted into the middle of this index? There's no space for new rows on any of these pages. And the logical ordering means that SQL can't just add new rows to any old page. So what you get is a page split. Half the rows on a page are moved to a newly allocated page, and then the row is inserted where it's supposed to go. This leaves us with two half-full pages and a page that's out of order. The physical order no longer matches the logical order. If I go along the file, my index key column value decreases. Lots of page splits along with deletes leaving holes where the rows used to be and your index starts looking like this. Here, my logical and physical order doesn't match at all. I have pages further in the file with lower index key values than pages earlier in the file. I have completely full pages. I have pages with gaps on them. I have pages which are mostly empty. This is a fragmented index. This is what index maintenance is intended to fix.

## Effects of Fragmentation

Fragmentation isn't the death null for an index. In fact, there's only two things that fragmentation causes. The first is a decrease in performance of large range scans from disk, emphasis large and disk. A range scan occurs when SQL Server reads a bunch of pages from the leaf level of an index. If this happens and the pages are not in memory, then the storage engine fetches the first couple of pages from disk and then kicks off what's called a read-ahead read. The storage engine assumes that because pages 1, 2, and 3 of the index have been asked for, the rest of the index is likely to be needed shortly. Instead of waiting for the rest of the pages to be requested, the storage engine issues multiple requests to the operating system in anticipation of the page that's been needed. The read-ahead process can request a lot of pages from the operating system in a single request, up to 64 contiguous pages. And there is the impact of fragmentation. If the pages of the index are all contiguous, then large amounts of data can be fetched in a single request. If the pages are not contiguous, then multiple read requests have to be made to fetch the data. This only happens though when you're doing read-ahead reads for larger indexes and only when the data is on disk, not in memory. You might notice this with an analytics type workload on servers that don't have enough memory, but you shouldn't really be noticing it on an OLTP workload. That tends to fetch single pages, not large ranges. The second effect of fragmentation is inefficient space usage. With a badly fragmented index, pages can be half full on average. This is a problem on disk as the database takes up more space than is strictly necessary, and it's even more of a problem in memory. When pages are read into the buffer pool, the page takes up 8 KB of memory, regardless of how much data is on that page. This makes memory usage inefficient, makes reads and writes from disk inefficient on top of all the existing problems of a larger-than-necessary database. All that said, fragmentation isn't a massive problem. There's a tendency in the SQL Server field to consider index rebuilds to be critical and any amount of fragmentation to be a disaster, and that's far from the case. In fact, it's extremely easy for index rebuilds to cause problems rather than fixing them. You should maintain your indexes, but it's not critically important that any amount of fragmentation be removed. It's really not that big a deal.

## Identifying Fragmented Indexes

If we're trying to identify our fragmented indexes, the DMV that we want to use is sys.dm_db_index_physical_stats. This DMV is unusual in that it is not a view into the internal memory structures of SQL Server, but instead it examines the index structures in the database. For that reason, it can be very slow on large databases. It has to read every single page of the index. Limited as a mode reduces that overhead as the index leaf pages don't get read, just the intermediate-level pages. You do lose some information that the DMV can return, but it's often worth the tradeoff. It still can be rather slow though. I limit investigations to only indexes with 1000 pages or more. Smaller tables don't suffer the effects of fragmentation because, below that, you're not likely to get those large range scans and because small, active tables are likely to be in the buffer pool most of the time, meaning you're not reading from disk anyway.

## Demo: Identifying Fragmented Indexes

In this demo, I'm going to go through the index physical stats DMV and show you the index properties, and we'll identify which indexes are fragmented and in need of maintenance. First thing I need to do is identify which indexes are fragmented enough to need maintenance. For that, I want the index physical stats DMV. I'm going to start by running it in limited mode. And with the WHERE clause commented out, that's going to return every single index in my database. One thing to note here is that the average page space used in percent column is null for every single row. That's because I'm running this in limited mode. In limited mode, the leaf level of the index isn't read, and so the average space on the page can't be computed. If I want that data, I'm going to have to accept the additional time required to run this in detailed mode. Quite a few of my indexes are very badly fragmented, 98 or 99%. But most of them have low page counts, 200 pages, 300 pages, and those are probably not a concern here. Index fragmentation affects large range scans from disk, and those indexes simply aren't large. So I'm going to uncomment the WHERE clause and look for indexes over 1000 pages and fragmentation over 30%. Why those numbers? Well, because somebody who worked on the SQL Server storage engine team 20 odd years ago was once asked for a recommendation and made an educated guess, and those were the numbers he guessed. They're decent. As guidelines go, they're frankly fine. You need a guideline, and I'll take the experience of someone who worked on the storage engine team and wrote this code over a wild guess of my own. The numbers are a decent starting point, but they are not numbers carved in stone. They're guidelines. They're a place to start. And once I add those filters, I've only one row. One index on my ShipmentDetails table is fragmented enough to need a rebuild, 99.9% fragmented and just short of 3000 pages. Let me run this again in detailed mode. With detailed mode, I've now got my page space usage column populated. And I can see that this particular index, the pages are 99% full. There's two reasons you might want to rebuild. One is the fragmentation, and one is if the page space usage is low because that's wasting space. In this case, I've got the high fragmentation, but my pages are mostly full. That's something that you need to consider and think about when you're choosing index rebuilds. Most people just go on the fragmentation. You can choose to ignore the fragmentation though and just go on the page space usage.

## Fixing Fragmentation

Now that we know how to identify indexes in need of maintenance, we need to look at what must be done to them. There are two options SQL Server has for fixing index fragmentation, the index rebuild and the index reorganize. Technically, you could fix fragmentation by dropping the index and recreating it, but that's a tad overkill. An index rebuild essentially does do a drop recreate on the index, but it has a bunch of optimizations built in to make it faster than dropping and then recreating the index. It is a full recreated the index though, done as a single operation. And as such, it can take quite a bit of time on larger tables and have quite a significant impact on the transaction log. Up until the latest versions of SQL Server, if you canceled an index rebuild part way through, the entire thing rolled back,

and you were left for the index as it was before starting. An index reorganize is a much lighter-weight operation. Instead of the entire index being recreated, the reorganize swaps out-of-order pages to get them back into sequence. It can also consolidate rows on two pages into one. It doesn't run as a single large operation, but rather as a series of small operations. And hence, it has minimal impact on the transaction log and retains the work done if it's canceled. It's not as thorough as a rebuild though. If we look at the options for the index rebuild command, there's a long list. This isn't even all of the options on an ALERT INDEX rebuild command. These are just the ones that affect the rebuild command. The ones that affect the properties of the resultant index I've left off. You can consult the Microsoft documentation for the full list. The PAD_INDEX and FILLFACTOR properties govern how much free space is left in the index page when the index is recreated. The default is 100, meaning that there is no free space intentionally left. The rows are packed as tight as possible. While this makes the smallest possible index, it's not a good idea in general. Having the index pages full means that the next insert or update that grows the row is going to cause a page split and start increasing the fragmentation again. There's no single value here that's ideal for fillfactor. Indexes that are on static tables can have a fillfactor of 100 without any problems. Indexes that get lots of out-of-order inserts or frequent updates should probably have lower fillfactors. I generally will start no lower than 90. And then, if I find the index is getting fragmented quickly, I'll drop the fillfactor down a little bit of a time until I'm happy with how often I have to rebuild that index. If pad index is off, which is the default, fillfactor affects the leaf levels of the index only. If pad index is on, it effects the higher levels of the index, the intermediate and root pages as well. I almost never turn pad index on. I find that fragmentation is mostly a problem with the leaf level. If I do find an index that's getting fragmented all the way up the tree, I'd probably turned it on. But it's going to be a case-by-case basis. SORT_IN_TEMPDB results in the sort space needed for the index being allocated from TEMPDB, not the user database. This does not mean that there's no space impact in the user database from the rebuild. The index is still recreated there. It's still logged in the user database. It's just the sort space that gets moved to TEMPDB. Consider this if you have limited space in the user database and a decent amount of space in TEMPDB, especially if they're on different drives, different I/O subsystems. ONLINE affects the type of locks taken during the rebuilt. By default, any index rebuild makes the entire table inaccessible for the duration. You can specify ONLINE = ON for the rebuild operation, and that removes those long-duration schema locks. It doesn't make the rebuild completely online, just mostly online. There are still short duration locks needed at the start and end of the operation. The WAIT_AT_LOW_PRIORITY option allows for those locks to be managed. You can set how long the alter index should wait to get those locks needed and define what it should do if it can't get them after the specified time has elapsed. None means that the rebuild operation continues to wait. Self means that if the lock hasn't been granted by the end of the specified time to abort the rebuild operation. BLOCKERS means to kill any sessions that are blocking the rebuild after the specified time has elapsed. RESUMABLE is a new option in SQL Server 2017. It allows for the index rebuild to be paused part way through and then resumed at a later point. MAX_DURATION is how long the index rebuild is allowed to run before it is automatically paused. This is excellent if you have, say, an hour a day to do index maintenance, but some indexes take longer than an hour to rebuild. You can specify that they rebuild for an

hour, pause, and resume the next day. There is an impact in space. The partially rebuilt index has to be kept until the operation is resumed and completed. Your paused index rebuilds can be resumed by running the exact same alter index statement again. Finally, MAXDOP overrides the server-level MAXDOP settings for the rebuild operation, specifying how many processer cores should be used for the rebuild. In comparison, the options for reorganize are very simple. Well, option, singular. There's only one option on altar index reorganize that applies to rowstore indexes. LOB_COMPACTION determines whether your large object data, that is your VARCHAR(MAX) and VARCHAR(MAX) VARBINARY(MAX), or XML data that is stored out of row is compacted or not. Compaction can reduce the space needed for lob data, depending on updates and deletes that have happened, but it won't always. With all that theory, I still haven't addressed how indexes should be maintained, and that's because there's no simple answer here. The commonly accepted industry standard answer is to reorganize indexes with a fragmentation between 10 and 30%, rebuild any with a fragmentation over 30%, and ignore any indexes with under 1000 pages in them. That is a decent starting position, but there are other considerations. How often can you rebuild indexes? How much maintenance downtime do you have? If the system is 24/7 with no downtime allowed, you're probably not rebuilding ever because even with the online option, locks are taken. Plus, recreating the entire index is a CPU and I/O-intensive operation and can impact the production load. Reorganize can be done with the load on the system. Rebuild, not so much. What kind of workload is the system predominantly running? An OLTP transaction processing system may not suffer any effects from fragmented indexes. An OLAP or analytics type workload may suffer terribly. Some tables might need their indexes rebuilding and some not based on the type of queries, especially if you have a hybrid workload. How much free space do you have? The log impact from an index rebuild will be the size of the index, plus log overhead in full recovery model. That can potentially be a massive amount of space needed in the transaction log. And if you've got availability groups or log shipping, that's log records that need to be copied to a secondary. In addition, there needs to be enough space in the data file for the new index because the old index is dropped at the end of the rebuild, not the beginning. Reorganize doesn't have these space requirements. And so if you're tight on space in the transaction log or the data file or you've got network limitations and need to ship the log records to a secondary, you might want to reorganize your indexes rather than rebuilding them.

## Demo: Fixing Fragmentation

So now that we've covered all of the theory, we're going to go and do some index maintenance on indexes that we previously identified as needing maintenance. I'll also talk about options for scheduling index maintenance from SQL Server's maintenance plans to PowerShell scripts. I've now identified the indexes or, in this case, index in need of rebuilding, so it's time to do the maintenance on it. I'm going to use simple ALTER INDEX rebuild here. It's really quick. This is a small table. 3000 pages is frankly tiny when you consider how big some production databases are, and so this should be really fast. If I go back and run my query, I now don't have any rows showing up as needing rebuilds. I should certainly hope not. A rebuild can fix 99% fragmentation without any difficulty. What I'm going to do

now is go back to the index physical stats DMV and look at this index specifically. It's returned three rows. Normally, you're not going to see multiple rows per index because of the filters. If you're filtering for 1000 pages and above and high fragmentation, you'll generally only see the leaf level. Because in this case, I've just filtered for the index name and I'm running detailed, I see all of the levels of this index. Level 0 is my leaf level, 3000 pages. Level 1 is my intermediate level. And level 0, consisting of a single page, is my root page. It's quite uncommon to see fragmentation of intermediate pages being high enough and with a high enough page count to trip the index rebuild threshold, but you might. If you are seeing multiple rows for a single index, you're seeing multiple levels of that index. Note that you only see that on detailed mode. Limited mode only returns information about the leaf level. I'm going to restore my database from a backup so that I could do that all again. This time I'm going to reorganize the index. In this case, rebuild and reorganize both wiped out all of the fragmentation. Reorganize did a slightly better job. You're not necessarily going to see the same thing in every situation. It's really going to depend on the database, the data, the layer of the tables, and a whole pile of other considerations. Generally, I'd recommend stick to the accepted guidelines of above 30% rebuild, between 30 and 10 reorganize, and ignore anything under 10, and ignore any index under 1000 pages unless you're finding that that doesn't work for you. If you really want to go down into the weeds, you can spend a lot of time looking at rebuild, reorganize, and playing around with details. The last thing I'd like to address is how to actually perform this maintenance. You can do it manually, but that's not particularly useful for production systems. Using SQL Server's maintenance plan is indeed an option in recent versions of Management Studio in SQL Server because in the later versions, the index rebuild maintenance plan has gained the ability to filter which indexes it rebuilds by page count and by fragmentation. In older versions, this wasn't possible. Still, even given that, I'd recommend not using maintenance plans. They're not that configurable. My preference is to rather use one of the index rebuild scripts I can find on the internet. My favorite is Ola Hallengren's index maintenance scripts, but there are others. What I do recommend here is not reinventing the wheel. Lots of people have put lots of time into creating index maintenance scripts. These are usually well tested, well documented. Rather use one of those than spending time writing your own.

## Summary

In this module, we looked at why you need to maintain indexes in the first place, looking at the effects of page splits on your indexes. We saw how to identify candidates for maintenance, and we looked at the considerations for rebuilding your indexes versus reorganizing your indexes. In the next module, I'm going to cover index maintenance again, this time for columnstore indexes. See you there.

# Maintaining Columnstore Indexes

## Introduction and Why Columnstore Indexes Need Maintenance

Hi, and welcome to this module on index maintenance for columnstore indexes. In the previous module, we looked at maintenance for rowstore indexes and saw why it was necessary and what the options were. In this module, we're going to do much the same for columnstore indexes, starting by looking at why they need maintenance because they are quite different from rowstore indexes. We'll see how to identify indexes in need of maintenance. And finally, we'll look at the options we have and considerations for maintaining columnstore indexes. Columnstore indexes don't suffer from fragmentation the same way that rowstore indexes do. They don't have keys at all, and hence there's no logical order for the index pages. Instead, rows are formed into rows groups of up to a million rows with no tree structures. The need for maintenance for columnstores is two-fold. When rows are added to a columnstore index, they're not added to the compressed segments. Rather, they're added to what's called the deltastore, which is a B-tree. These deltastores show up in the metadata as open rows groups. When a deltastore reaches a million rows, it's not compressed immediately. There's a background process called the tuple-mover, which takes those rows groups and compresses them. But it is possible when doing lots of modifications to have open rows groups being created faster than the tuple-mover can compress them. The second thing that leads to columnstores requiring maintenance is that rows can't be deleted from a columnstore index because they're part of the compressed row group. Instead, the row is marked deleted in what's called a delete list. And when any query reads the columnstore, those deleted rows are read and then removed from the results. If you've got a lot of deleted rows in a row group, that can add up to a fair bit of overhead to the reads. You could be reading a million rows to return 200,000 if most of the rows are deleted. Oh and updates, they're processed as a delete/insert pair, which means they add to the deleted list, and then they add to the deltastore. So these are two of the three things that lead to a columnstore index needing maintenance. Lots of uncompressed rows groups will slow down reads for the simple fact that they're not compressed, and a lot of columnstore performance gains are from that compression. Hence, if the tuple-mover is not keeping up with the growing deltastore, that index needs maintenance to force the compression of those rows groups. If there have been lots of deletes, then those logically deleted rows need to be removed from the rows groups completely, rather than just being marked as deleted. This also requires an index maintenance operation. Finally, it's possible to get rows groups with under the ideal number of rows. This can come from memory pressure when creating the columnstore or from dictionary pressure. I'm not going to go into detail on these. There will be a link in the last module to Nico's blog, which has all the details of all of these and more. If this does happen, you probably want to consolidate the rows groups because a lot of the benefit to the columnstore comes from

compression, and that compression is optimal with a million rows in the rows groups. That consolidation is going to be done by an index maintenance operation. Most of the metadata around columnstore index maintenance needs is found in the sys.columnstore_rows_groups system view. This query is looking for indexes with open rows groups, i.e. deltastores. Ideally, you don't want to see more than a couple of these open row groups in any columnstore index. A similar query shows the number of deleted rows per index. My rough guideline here is that anything more than 10% of rows deleted needs attending to, but that will vary per system. If you see performance degradation on column stores from a lower percentage of deleted rows, then base your index maintenance on that threshold, not on mine.

## Demo: Identifying Columnstore Indexes in Need of Maintenance

I'm going to do a quick demo here. Firstly, insert a large number of rows so that we can see the multiple uncompressed row groups. Then delete a portion of the table and have a look at the system views and see how the deleted rows appear. Before I started this demo, I updated about a quarter million rows in a table that has a column store index on it. This index is hence going to have some deleted rows and some rows in an uncompressed row group. Let's have a look and see how that all shows up. The sys.column_store_row_groups view has one row in it per row group per column store index. I only have one columnstore index in this database, and so that's the one you see. The query does some aggregations, and hence we can see how many row groups we have in the particular states. At this point, we've got one compressed row group and one open row group. The open row group is our deltastore, and our compressed row group has 219,000 rows deleted from it. If I take a slightly different query that doesn't have the aggregation and look at just the row groups which are not compressed, I can see the we've only got one at the moment and it's open. Okay, this is our deltastore. Let's see if I can break things a little bit. I'm going to do some inserts into this table, querying from this table. So I'm just inserting a lot of data, and I'm going to run that 10 times. The background tuple-mover is not going to keep up with this. So by the time this is done, I should have a large number of uncompressed row groups. The inserts took quite a while to run, but they're finally all done. Now if I run the query I had initially, I can see that I've got row groups in four different states. I've still got my compressed row groups, I've still got my open row group, and I've only got one open row group. That's my active deltastore. I also have 38 closed row groups. These are not compressed. These came about when the deltastore reached a million rows. It was then closed, scheduled for compression, and another deltastore was created. I also have 11 tombstone row groups. These are leftovers after the tuple-mover has compressed our closed row group. So the tuple-mover has been able to compress 11 row groups, and it has 38 more to compress. This index is definitely in need of some maintenance at this point. And in the next demo, we'll see how.

## Effects of Rebuild and Reorganize on Columnstore Indexes

Now that we've got the indexes that need maintenance, it's time to look at our options for fixing them. We have the same two options for columnstore indexes as we had for rowstore indexes, rebuild or reorganize. They do very different things to columnstores however. If we rebuild a columnstore, it recreates the index completely. Because it recreates the index completely, there will be no open uncompressed row groups at the end of the rebuild, only the compressed row groups. Again, since it's recreating the entire index, there will be no deleted rows left in the columnstore index either. The problem with rebuilding is that it's memory-intensive, and it takes a lot of time, and columnstore indexes tend to be built on larger tables as they have minimal gains on small tables. So that rebuild is going to take quite some time and a large amount of log space. It can be done a partition at a time on partition tables, and in that case, it's likely only a subset of the partitions will need rebuilding, but it's still pretty intensive process. And if you are suffering from memory pressure or dictionary pressure resulting in small row groups, a rebuild may just result in more small row groups. You still have that memory pressure problem. Reorganize is probably what you're going to be doing most of the time. For a columnstore, there are no pages to shuffle back into order. Instead, the reorganize works on one or two row groups at a time, cleaning them up. There are three things that reorganize does. It compresses all of the deltastores, all of the noncompressed row groups. This is ideal after a large data modification process that's left the whole pile of uncompressed row groups. Bulk loads should create the row groups compressed directly, but there are a lot of other operations that won't. If you've got some import process or data processing job that leaves uncompressed row groups, it may be a good idea to add a reorganize step to the job after the processing is complete. The second thing that reorganize does is to remove deleted rows if more than 10% of the rows in that row group are marked deleted. This was a change added in SQL Server 2016. In earlier versions, reorganize didn't affect the deleted rows at all. The last thing that reorganize does is to combine two row groups in to one if their combined number of rows total less than a million. Again, this is a change in SQL Server 2016.

## Demo: Maintaining Columnstore Indexes

So in this demo, I'm going to look at a couple of columnstore indexes that are in need of maintenance, and I'm going to show the effects of rebuilding them and the effects of reorganizing them. In the previous demo, we identified this columnstore index as needing maintenance. I'm going to start by rebuilding it and having a look at how the rebuild does. I've already taken a backup of this database before the rebuild. So once we've had a look at how the rebel did, I'll restore it from backup, and we can looked reorganize. For now, let me rebuild it. It only takes 45 seconds, but then this is not a particularly big columnstore index. And there's no concurrent usage. This is my desktop PC. As with rowstore indexes, these rebuilds can take quite a bit of time on larger tables. Before I ran the rebuild, there were 38 uncompressed closed row groups and 11 tombstone row groups in the index, along with the compressed row groups and the open row group. After the rebuild, I have 63 compressed row groups with 0 deleted rows. The rebuild recreated the columnstore index completely. I have no closed row groups. I've got no tombstone row groups. I've got nothing for the tuple-mover to be doing. And I've got no deleted rows. This has done a complete recreation of the

columnstore index. I have 61 million rows and 63 row groups, meaning we've come reasonably close to the million rows per row group on average, but we are slightly under. Let me restore that database from backup and see how reorganize does. The reorganize took longer than the rebuild did, 2 minutes, 20 seconds to be exact. If we look at what it's done, we've got 58 compressed row groups and one open row group. The open row group is again our deltastore. This isn't a recreation of the columnstore index. I still have some rows in my deltastore. The reorganize has done better than the rebuild did in terms of full size row groups. I've got 58 row groups with 60 million rows as opposed to the 63 row groups I had after the rebuild. So I'm just over a million rows in each row group on average, which is what we want. These are full row groups. I still have the tombstones, but those will be cleaned up over time. And I've still got my deltastore with some rows in it. But in terms of fixing the deleted rows and compressing the closed row groups, the reorganize has done a pretty good job cleaning up this index.

## Summary

In this module, we looked at the maintenance needs for columnstore indexes. We looked at the impact of open row groups and deleted rows on columnstore performance, and we looked at what rebuilding the columnstore index does versus reorganizing it. In the next module, I'm going to have a look at statistics maintenance, including what statistics are, why maintenance is needed, and how we maintain these statistics objects. See you there.

# Maintaining Statistics

## Introduction and What Are Statistics?

Hello, and welcome to this module on optimizing statistics. This is part of the Optimizing Indexes and Statistics course for Pluralsight. In this module, I'm going to look at statistics, starting with what they are and what they do. Not a lot of the detail, just an overview. Once we've looked at why they're important. I'll show you what happens when they're run. We'll look at the need and options for statistics maintenance, and I'll end the module with some guidelines around statistics maintenance based on what I've seen over the years. So let's start by looking briefly at what statistics are. At a high level, they're aggregated data about the data in the tables. They're created like indexes are on a column or set of columns. Every rowstore index has an associated stats object, and you can also have stats objects not associated with indexes. There's two pieces of information in the statistics object, which are critically important. The first is a measure of the uniqueness of the left-based subsets of the columns. This is expressed as a ratio, unique values in the columns divided by the total number of rows. The highest this can be is 1 for a unique set of columns, going down as the columns gets less unique. The second is information about the distribution of data in the leading column. This is stored as a histogram with a maximum of 250 steps. Each step stores of value from a column, the number of rows that have that value, and the number of rows that have values between the previous histogram steps value and this one. What's important, especially in the context of this course, is that the stats objects are not kept up to date as the data in the underlying table changes. This is different to indexes, which are always in sync with their tables. Stats objects are used by the query optimizer when it generates execution plans. To generate good plans, the optimizer needs to have some idea of how many rows a query will affect, and it needs to know that before the query runs. It uses the stats object for that. The stats are used to estimate the number of rows the different operators in the plan will affect and, in some cases, how many times an operator will execute, and also to estimate the total size of the data being processed. These estimates are then used to select the cheapest plan based on what operators are optimal for the number of rows estimated. Estimates are also used to calculate how much workspace memory will be needed to execute the query. If the statistics become incorrect, usually as a result of data changes, there's a number of things that can happen as a result. Slow queries is the simplest. With incorrect statistics, the optimizer generates plans that thinks will be optimal, but which are not, wrong operators for the row count, wrong indexes, insufficient memory grants resulting in spots to TempDB, all of those are common effects. I've seen an important morning batch job go from running to completion in around 40 minutes to not finishing after 5 hours because the statistics had got just wrong enough the optimizer to make a bad choice of path. Related is high CPU usage. Operators that are optimal for small numbers of rows often scale quite badly and use lots of CPU when they're forced to process large numbers of rows. Similarly, you can have excessive I/Os. If the optimizer underestimates the row count, it may assume that a full table scan is an efficient way of getting the data. But if the

table is much larger than estimated, that can have nasty effects on memory usage and I/O load. Finally, data sizes are used to estimate how much memory will be needed to execute the query. If this estimate is wrong, the interim results during query execution may need to be written to TempDB during processing, potentially many, many times.

## Demo: Reading Statistics and Their Effects on the Execution Plan

So in this demo, I'm going to have a look at some statistics objects and show you what data is in them, and then we're going to intentionally get some statistics wrong and show the effect on query plans. To see what data a statistics object has, we need to use the DBCC SHOW_STATISTICS command. This takes a table name and a statistics name. That name is the name of an index. And in this case, I'm going to look at the statistics associated with that index. SHOW_STATISTICS, by default, returns three result sets. The first is the overall statistics for the column that that statistic is built from, the second lists the densities of the left-based subsets of the columns involved in the statistics object, and the third is the histogram. Interesting information in the first one, we have when the statistics were last updated, we've got the total number of rows in that table, and we've got the number of rows that were read to generate the statistic object. Please ignore the density column in there. It's not what it says it is. It's actually wrong. If you look at the second one, we've got a measure of uniqueness for the columns. This index is built on TransactionDate. And so since it is a nonclustered index, it gets the clustered index key added to it, making it effectively an index on TransactionDate and TransactionID. And there you can see how unique those two sets of columns are. Firstly, TransactionDate as the left-most column, then TransactionDate and TransactionID combined. The density is 1 divided by the number of unique values. The second one is unique completely because TransactionID is the primary key. The density of the TransactionDate is slightly higher. You can use this to work out on average how many rows match a particular value in this table. If we look at the histogram, the RANGE_HI_KEY is a value from the table. This creation of the statistics selects interesting data points from the table. RANGE_ROWS is how many rows between that value and the one before it. EQ_ROWS is the number of rows matching that value exactly. DISTINCT_RANGE_ROWS is the number of unique values between that hi key and the value before it. And AVG_RANGE_ROWS is for that interval on average how many rows will match a particular value. And if we looked at the end of the histogram, we can see the at the last value recorded there is the 2nd of January at 21:41. That is going to be very close to the last value in the table at the time the statistics were created. So let me take a query, which filters on TransactionDate. The number of rows returned by that filter is going to determine the shape of the execution plan here. If I put in a value that returns about half of the table, I get about 46 ms of CPU time and a couple 100 reads from that query. And if we look at the execution plan, we've got two clustered index scans and a hash join. The reason the index on TransactionDate isn't being used is because it's not covering, and the optimizer has figured out that the cost of looking up the rows from the clustered index is too expensive, and a nested loop join that would be used in that case is also too expensive, and so it's gone for scan and a hash join. If I instead put in a value very close to the highest one in the table and the FREEPROCCACHE is making sure that we have new plans

every time, we get a very different shape of plan. We get an index seek, we get a key lookup, and we get two nested loop joins. That's because for this parameter value, the statistics show that there aren't going to be very many rows returned. And so operators which are good for small numbers of rows, loop joins and key lookups, are indeed find here, and that's what the optimizer has used. Let's see if we can break things. I'm going to turn automatic statistics updates off on this database and then update half of the table, setting the TransactionDate to 6 months higher than it was. This should push a good few 1000 rows out past the highest value in the histogram. If I go back and run that query again, we've got the same plan as we saw before for the second of January, index seek, key lookup, nested loop join. The optimizer has chosen it because it still thinks that that's a good plan because the statistics haven't changed. But this time, it's returning 6630 rows rather than just a handful. And if we look at the properties of that index seek, we can see estimated number of rows 10.28. Actual number of rows 6630. This is the effect of the incorrect statistics. The estimation was based on the stats. The actual number of rows encountered at execution time was quite a bit higher. If we look at our execution characteristics, we have much the same CPU time as we had for the original query with the hash join and the table scans. But our reads, the number of pages we process to do that query, is much, much higher, even though we're returning only half the number of rows that we did with that initial query. That's the one that filtered for all dates greater than June. And if we go back and have a look at statistics object, the highest row in the histogram is still the 2nd of January at 21:41.

## How Statistics Get Updated

Now it's not all doom and gloom. There are built-in processes to automatically update statistics. And in many cases, these are adequate. An automatic statistics update is triggered after a certain number of values have changed in the underlying table in the columns the statistics are built for. This threshold is a sliding scale in recent versions of SQL Server. In older versions, it was a straight 20% of the rows in the table having to change, and that was often far too many changes in larger tables. In more recent versions, the larger the table is, the lower the percentage of rows is that has to change before an automatic update is triggered. A stats update doesn't run immediately. Once the statistics are invalidated, it's the next query that needs those stats that pays the price and has to wait for the statistics to be updated before it can run. There is a database setting, Allow Auto Update Asynchronous, that allows the query that triggered the stats update to run and have the stats being updated by another thread. It does mean that the triggering query runs using stale statistics though. I've never seen a case where that was needed. But if you do have a system where the automatic stats updates are causing query delays, consider turning that option. On larger tables, a read of all the data to generate new statistics would take quite a lot of time and could displace quite a bit of the buffer pool. And so the larger the table is, the smaller the portion of the table is that is read. Small tables will likely have a stats update read all of the values in columns, while large tables get a sampled update. The larger the table, the smaller the sample, by default. This can be a problem if the table has data skew. It's possible for sampled update to miss important values in the table. This can result in all of the same problems that out-of-date statistics cause. The

automatic update is not the only option that we have for keeping stats updated though. Of the three options that we have for keeping statistics up to date, automatic updates are probably the most commonly used, and it's one that will work fine in most cases. Smaller tables or ones not frequently changed will likely be fine with the automatic update. There's also index maintenance. When an index is rebuilt, not reorganized, but rebuilt, the stats associated with that index will be updated with a full scan. The entire column or set of columns are already been read to rebuild the index, so it's no extra work to update the statistics at the same time. Now this is only for statistics associated with indexes, not your independent stats objects. The third option is manual updates. That's going to be for when the previous two methods don't suffice, either because index maintenance hasn't been done often or because the stats are not part of an index. Most of the index maintenance tools that I mentioned previously also have statistics maintenance capabilities. And if you are using one of those index rebuild scripts, they'll probably be handling your stats updates just fine. The syntax for manual statistics update is much simpler than for index rebuilds. You can specify whether to do a full scan update, a sampled update, and if so with what sample, or to use the last sample rate if you specify resample. If you are taking the time and effort to do manual updates to stats, I recommend full scan, unless that's impossible due to your time limitations. NORECOMPUTE disables automatic stats updates on the statistics object. This probably is not a good idea in most cases, but I can think of some scenarios on larger tables where the auto updates are sampled, and you don't want the auto update kicking in because it's going to do a sampled update of the statistics, and you know those are not adequate. Definitely not something to use by default, but there are places where it's valuable. INCREMENTAL computes the statistics of the partition level and then merges the results to form a global statistics object for the table. It defaults to off, and it throws an error if it's used on a nonpartitioned table or index.

## Demo: Updating Statistics

In this demo, I'm going to go through the various options for updating out-of-date statistics and show you the effects. In the previous demo, we saw the effects of out-of-date statistics. Now let's see if we can fix them. I'm not going to turn the auto updates back on. That's too easy. Let me rather show you how to do a manual statistics update. We've seen in previous modules how to do the index maintenance, and index maintenance, specifically index rebuilds, do update statistics. But if you're not going to do index maintenance, there has to be another way to manage the statistics, and that's with the UPDATE STATISTICS command. UPDATE STATISTICS takes the table name and optionally the index or statistics name that you want to update the statistics on. If that's left out, all the stats on the table are update. Finally, you can say with sample and give a percentage or with full scan. Leave that out and the default sampling is applied for that table size. In this case, I'm going to do a full scale update. If I look at the statistics object again, the Updated column in the header has changed to just now, reflecting the update of the stats, and the histogram has also changed. So let's go back to that query that we had problems with earlier. Without changing the query, I'm going to run it. The stats have been updated. We've got a different execution plan. We've no longer got the

nested loop joins. We've no longer got the key lookups. Instead, for this number of rows, the optimizer has calculated that a merge join along with two clustered index scans and a sort will be the most efficient way of executing this. And if we look at our execution characteristics, our CPU time is down, down to 30 ms, and our reads are back down to a couple of 100 from the tens of thousands we had with the bad plan. If we go back and have a look at the histogram in our stats, we can see the at the highest value in the statistics is now the 2nd of July, reflecting the results of that update that I ran earlier. You can do statistics updates with maintenance plans. But like with index maintenance, I don't recommend their use. There's just not enough configurability on the maintenance plan. The only options we have with the update statistics task are whether we want to update all stats, column stats, or index stats and what scan percentage we want. There's no ability to only update stats which are out o -date, only update stats with a certain number of changes, and many of the statistics maintenance scripts that you'll find have those abilities.

## Summary

The question then remains, how should you do statistics maintenance? Based on systems that I've seen over the years, my main recommendation would be please do some. Some form of statistics maintenance is better than none. And while the auto update works well enough on smaller and more static tables, it often doesn't on larger tables or more frequently modified tables. It's not harmful in the majority of cases to update all statistics with full scan whenever you have the maintenance time. This is the opposite to index maintenance. I find index maintenance is often overdone. Statistics maintenance, I find, is underdone. If you have the time, it's better to update stats more often and rebuild indexes less often than the other way around. Stats updates don't have huge effects on the transaction log, although they do read a lot of data, and hence it's quite difficult to overdo the stats maintenance. It is possible though. For example, I don't recommend updating statistics hourly, although I have heard of a case where updating the stats of one table every hour was indeed needed. The main side effect of updating stats is invalidating cached execution plans, but often that's a good thing. It means that the new plan will be based on up-to-date data. But again, if you're having cache displacement problems or plan cache churn, you might want to turn your stats maintenance down or schedule it for specific times. One thing to be really careful of though is that updating stats with a sampled update after you've done index maintenance is a waste of time. It's worse than a waste of time, in fact. The index rebuild updates the stats associated with that index with full scan. If you then do a sample stats update, you end up with worse statistics than the rebuild would have left. So you've wasted time and got a worse result. Most of those index and statistics maintenance scripts I discussed previously avoid doing that. And again, here, I recommend not reinventing the wheel. Don't write your own statistics maintenance scripts. It's really not worth it. So in this module, we looked at what statistics are and how the query optimizer uses them. We had a look at the query plan and saw just how bad the out-of-date statistics can affect query performance. We discussed the maintenance options, automatic updates, index maintenance, and manual statistics updates, and then we had a brief chat about the guidelines for when to update statistics and how often. In the next module, I'm going to wrap up the whole course

and reference a few additional courses, blog posts, documentation pages, and other resources that I think will be a value to you. See you there.

# Summary and Further Reading

## Course Summary

Hi, and welcome to this final module in the Optimizing and Maintaining Indexes and Statistics course. In this module, I'm going to summarize everything that we've covered in the course, and then I'll reference some related courses, as well as some documentation pages and blog posts for further reading. The sections that we went through in this course were missing unused and redundant indexes, rowstore index maintenance, columnstore index maintenance, and statistics maintenance. In the missing index section, we started by looking at the missing index DMVs. We saw where the recommendations came from, looked at some limitations of the missing index DMVs, and saw how to create indexes from them and test them out. We then went on and had a look at the Database Tuning Advisor. This is a separate tool to SQL Server that does a much more comprehensive workload analysis, but it's also got some limitations. We looked at the options for generating workloads, went through the analysis process, and had a look at some limitations and cautions around Database Tuning Advisor, mostly around its habit of seriously over-recommending indexes. The main thing I'd like you to take away from this section is that it is critically important if you're using these tools to test and evaluate the indexes that are suggested by them before you go and create them on your production server. They are not perfect. They are not flawless. And you absolutely have to test their recommendations. We then went on to look at unused indexes. This is tracked by the index usage stats DMV. This is a memory-only DMV, so the data is lost when SQL Server restarts. This tracks any access to the index, whether it be from reads or from updates. You can use this to identify indexes that have never been used. You can use this to identify indexes that have been modified, but not read. There's a lot of information you can get out of this DMV that will help you tune the existing indexes you have in the system. The important thing here is that you must monitor long enough to catch usage for a full business cycle. Because this is a memory-only DMV and the data is lost when SQL Server restarts, if you aren't storing this data somewhere else, you could easily end up losing information about index usage for events like month ends or year ends or other occasional processes. Dropping indexes that appear to be unused, but are only used during these critical times can be really, really painful. So again, here, the guidance is test carefully. The data from the DMV is a starting point. It's not the final answer. You do have to do a lot of additional analysis, and you do need to test carefully before dropping indexes in production. We went on to look at redundant indexes. These are not generated from data changes or schema changes. They are often generated by someone being less than completely careful about creating indexes, sometimes with the missing index DMVs or the Database Tuning Advisor. The characteristics which identify a redundant index is when you have two or more indexes with the same key columns in the same order. The include columns don't matter here because you can consolidate them. It's

the key columns that are important. Or you can have one index's key being a left-based subset of another indexes key. Be careful here. You want to make sure that the columns are left-based subset and the columns in the same order. If you've got an index on column 1, column 2 and another index on column 2 alone, that's not a redundant index situation. If you had a second index on column 1 alone, it would be, but not the second column. When you're looking at consolidating redundant indexes, the key columns overlap, and then you can combine the include columns. This is why I don't look at the include columns when I'm identifying whether indexes are redundant or not. And again, test these very carefully before you make these changes in production. It's very easy to make a mistake or to have a situation where a smaller index really is useful. They're rare, but they do happen. From there, we went on to rowstore index maintenance, looking at the index physical stats DMV in order to determine our fragmentation. We spent quite a bit of time looking at fragmentation, how it comes about, and what implications it has for your database. The general guidelines for index maintenance are rebuild any indexes over 30% fragmentation, reorganize between 10 and 30, and ignore anything under 1000 pages because it's just too small to matter. These are general guidelines though. Your mileage may vary. And if you're seeing something in your production system that contradicts these guidelines, please go with your recommendations, not mine. The thing to keep in mind here is that index maintenance can be overdone, and it's not as big a deal as a lot of people make out. So if you're not rebuilding indexes every night, that's probably fine. If you're rebuilding them every week, that's probably find as well. Don't panic over 10 or 15% fragmentation, even in large tables. The maintenance needs for columnstore indexes don't come from logical fragmentation because they don't have any of that. Instead, it comes mostly from deleted rows and large numbers of closed rowgroups. Deleted rows lead to a need for maintenance because they're not removed from the columnstore when the delete happens, but instead are marked deleted in a deleted list. Large numbers of closed rowgroups come from when the tuple-mover is not keeping up with inserts. The tuple-mover is a background process, which compresses the rowgroups. Closed rowgroups are not compressed, and so if you've got a lot of closed rowgroups, you're losing out on a lot of the benefit of the columnstore index. The system view that you want here is sys.column_store_rowgroups. This is not a DMV. It's instead a system view. And this has one row in a per rowgroup, per columnstore index. So you can use this view to identified columnstore indexes that have rowgroups with large numbers of deleted rows or columnstore indexes that have large numbers of closed rowgroups. Finally, we looked at statistics maintenance. The statistics objects are summarized data about the data in the table, and the important thing is that they are not kept up to date as the data changes. Instead, there's an automatic update process that's triggered once there's been a certain number of changes made to the column underlying the statistics. Those automatic updates are often good enough, especially if you've got small tables or infrequently changed tables. But in the cases where it's not, you should be doing some form of statistics maintenance in order to keep the stats up to date and ensure that you've got good execution plans for your queries. The UPDATE STATISTICS command is what you want here for updating statistics manually. While an index rebuild does update stats, if you just need stats maintenance, rather just do the stats maintenance and don't rebuild the index as well.

# Further Reading and Additional Courses

Going forward, there's some other Pluralsight courses that I'd recommend, ones that either I've mentioned in this course or ones which contain further information or related topics. To start with, Paul Randal recorded an absolutely exceptional course on index fragmentation. Paul Randal used to work on the SQL Storage Engine Team, and he knows the Storage Engine backwards. This course is about 8 hours long, and it goes into a huge amount of detail. I spoke a few times about testing workloads. I did a course on SQL Server's distributed replay where I show how you can use that to simulate production workloads. That may be very useful for your testing. And finally, I talked about Query Store a few times. If you're not familiar with Query Store, I highly recommend Erin Stellato's course on Query Store where she goes through what Query Store is and how to use it. Moving outside of Pluralsight, there's some documentation and blog posts, which are interesting and relevant. Firstly, there's a Microsoft documentation page on indexes. This is mostly around creating indexes, but there's a section on maintenance as well. I've mentioned Ola Hallengren's Index and Statistics Maintenance a few times. You can find his entire maintenance solution at this URL. Highly recommended, used by a lot of people in a lot of companies. For columnstore indexes, Nico has written a huge number of blogs, it was over 100 at this point, on various aspects of columnstore indexes. This is his post on what would cause small rowgroups when an index is created or rebuilt. He goes into the memory and dictionary pressure here. And then, going back to the Microsoft documentation, their page on columnstore index maintenance is incredibly detailed and has a huge amount of information on it. And finally, on blog posts, Paul Randal, author of the course that I previously mentioned, has a blog, and one of the sections with his blog is indexes. He talks a lot about index maintenance on this. So if you're looking for more information, you can't go wrong with Paul Randal's course. That's it for this course on index and statistics maintenance. I do hope you've enjoyed this, and I hope you've learned something of value to you. Hope to see you in a future course. Goodbye