Course Overview

Course Overview

Hi everyone. My name's Gail Shaw. Welcome to my course on Designing and Implementing SQL Server Database Indexes. I'm a technical lead in the data solutions division at Intellect Software based in Johannesburg, South Africa. Good indexing is critical for well- performing databases, but very few databases that I see in production have anything resembling good indexing. Learning how to create indexes to support your database workload will allow you to ensure that your applications are performing to their peak. In this course, we're going to have a look at several types of indexes that SQL Server has. We're going to look at how to choose a clustered index and what the tradeoffs are for various choices there. We'll look at nonclustered indexes, indexes which you create to support your workload. We'll look at indexed views and see where they're useful. And we'll look at columnstore indexes, how they differ from traditional indexes, and where they excel. By the end of this course, you'll be able to create effective indexes for almost any SQL Server workload. This is an introductory course, so no prior experience in indexing is required, but you should be familiar with SQL Server table design and basic queries. Familiarity with execution plans is a bonus. I hope you'll join me on this journey to learn all about indexing with the Designing and Implementing SQL Server Database Indexes course, at Pluralsight.

General Indexing Guidelines

Course Intro

Hi, and welcome to this course on Designing and Implementing SQL Server Indexes, from Pluralsight. Before we dive into the details though, why do we create indexes in the first place? There's two main reasons, or should I say, there's two main reasons why we should create indexes, not two main reasons why indexes get created. The two reasons that we should be creating indexes are one, to enforce uniqueness on data. Indexes are used to enforce primary key and unique constraints, which

enforce uniqueness upon a column or a set of columns. This is part of database design, and I'm not going to go into this much in this course. In fact, I'm going to mention it once and only once in one of the later modules. It is important, though, when you are designing databases to ensure that you've got the appropriate unique constraints created to ensure the integrity and correctness of your data, but that's not performance, and this course is about creating indexes for performance. So the second reason is indeed to improve the performance of queries. I do a lot of performance tuning for client systems these days, and I generally see three main causes for performance problems, bad database design, bad indexing, bad queries. This course isn't going to help you with the first or third of those, but it is going to help you identify the indexes that you should be creating for your workload so that you've got well-performing queries. One thing that is critically important, however, is to understand the indexes which we create for performance are determined by the workload that runs against the database. You cannot create indexes based upon the table designs alone or the data alone. There's stories from conferences about people bringing database designs to somebody and asking how would I index this? And the answer is I have absolutely no idea how you'd index that. What queries run against these tables because the queries that run against the tables are what determine the indexes that you want for those tables because we create indexes to support the workload, not for the sake of creating indexes. There are two possibilities that you might have when you're looking at creating indexes for a system. Either you've got a brand-new system, one that is not in production that has no users using the system, the second is that you might have an existing system that is not performing well that you need to add indexes on. How you approach the indexing for the two is slightly different. If you've got a new system, the first thing you want to do is create indexes to support your unique requirements. Which columns or which sets of columns unique? Those need to have primary key unique constraints or unique indexes created on them. Create indexes on your foreign key columns. This is the foreign key columns on the child tables. These may not actually be used. They're not necessarily indexes that you'll be keeping for long, but it's a good idea to start with. Create them, but keep in mind that you might change them later. And then create indexes to support expected common queries, but be careful here because what the developers of a system think will be common queries are not always what the users actually use the most. I've got a lovely story from back when I worked at one of the banks. The users thought that the contact list would be the most used part of the system and hence, the queries behind the contact list would be the most important to index. The developers

thought the call logging system would be most important, that that would be used the most and hence, we should focus our indexing effort on that. Once we analyzed the system, however, we found that the custom security queries were actually found more important than either of those. And by indexing the custom security, we got massive performance improvements. So speak to the developer, speak to the testers, speak to the business and figure out which queries are expected to be running frequently, and be prepared to change these later. Once the system is in production, once it's in use, once the use is actually doing things on the system, you may find that the usage pattern's different from what you expected, and so you need to change indexes to support the new usage patterns. If you've got an existing system, what you probably want to do here is identify the most resource-intensive queries, the ones that are using the most CPU, using the most memory. You'll also want to identify the most important queries. Which queries are critically important must execute in a short period of time or perhaps are run by the CEO in the corner office. You don't want to disappoint the CEO after all. Index to support those. Create your indexes to support those important or resource-intensive queries, and not all of them. Generally when I'm doing performance tuning, I'll create indexes for five or six queries, no more. And then investigate regularly and repeat this process as often as necessary to keep the system performing well. That was a high-level look at why we index in the first place. Now let me quickly go through what the rest of this course will cover. First, we're going to look at overall index architecture, how indexes are used by SQL Server, and how those uses appear in the execution plans for queries. We're going to look at clustered indexes and whether we should create those to support queries or to organize the table and look at the tradeoffs inherent in that choice. We're going to look at nonclustered indexes and how you index various types of predicate. We're going to briefly look at index views. They're interesting, but unfortunately, they're very limited and I find them to be seldom useful in production systems. We'll look at them anyway because they are useful at times. And then we'll look at columnstore indexes, a new, I say new, but they've been around 7 years, a new type of index used for analytics queries that promise massive performance improvements. I hope you'll join me for the rest of this course on SQL Server indexes to see how they work, how you should create them, and how you can make your system blazingly fast. See you in the next module.

Understanding Basic Index Architecture and Index Usage

SQL Server Index Architecture

Hi, and welcome to this module on Basic Index Architecture. In this module, we're going to take a high- level look at how index is constructed, we're going to see how SQL Server uses those indexes, and then we're going to have a quick look at SQL Server execution plans and see how the various index operations show up. Let's get straight into architecture. When we talk about indexes in SQL Server, we talk about index trees. SQL Server stores data within the data file in pages that are a fixed 8KB in size. Most size and space measurements that you'll see coming from various system views and functions return sizes in a number of pages. These pages are then stored in a tree structure. For the computer science people, the specific type of tree is a balanced tree or a B-tree. The index key is the column or set of columns that the index is built upon, and the index is logically ordered by those columns. So let's build an index. Say we have this set of data, its pairs of transaction types and amounts from the transactions table, and we want to index on this data. The first thing we need to do is sort the data by the index key, in this case, transaction type and amount. With the data sorted, we can then build the tree structure. First, divide the data into 8KB chunks ordered by the index key. Now, in a real system, we'd fit a lot more than five rows of data onto a page, but to keep things simple, let's pretend that this is all we can fit. This is called the leaf level of the index. An index will always have one or more leaf pages. Next, we need to build the level above. We do that by taking the first row, the row with the lowest key value, from each page and we add those to new pages. In this case, since I have eight leaf pages and can only fit five rows a page, we have two of those pages in the next level up with four rows in them each. Since there's more than one page at this level, it's called an intermediate level, and we still need one more level. An index will have 0 or more intermediate pages. Had we been able to fit the first row of each page in the leaf level into a single page, we wouldn't have an intermediate level at all, but since we couldn't, we have a single intermediate level with two pages. Again, we take the first row of each page to build the level above. This time, it's two rows, which do fit onto a single page, and so we get the root level of our index. An index will have 0 or 1 root pages. If

the leaf consists of a single page, there's no need of a root, but in all other cases, we will have one and only one root page. This is the general form for our index tree. We do, as you've probably noticed, like to build our trees upside down.

How SQL Server Uses Indexes

There are three ways that SQL Server uses indexes. The first one is they can perform a seek operation, a navigation down the index tree from the roots to the leaf in search of a value. This could be a single row that's been searched for or it could be the start or end of a range of rows. The second is a scan. This is a read of one or more pages in the leaf level without navigating down the index tree first. The third is one that I'm not going to go into detail on now. It'll come up in a later module. This is a key lookup. It is a single-row seek to the clustered index. I'll be talking about clustered indexes in the next module and key lookups in the module after that. For now, just be aware that it does exist. So let's see practically with an index tree how SQL uses its indexes. We'll start with a seek operation. For this, we need to be looking for a particular row or a particular value. Let's say we're looking for the row C 128.86, that is a transaction type of C and an amount of 125.86. Each non-leaf level contains the lowest row of each page below it. In our case, the two rows on the root page have transaction types of A and B and hence, we know that all rows with a transaction type of C will be found by navigating down the second link. The next page we encounter is an intermediate page, hence we need to repeat the same process. The row we're looking for is larger than B, 297.45. It's larger than C at 65.23, but it's smaller than C 301.01 and hence, the row we want will be found on the second of these pages. This page is a leaf level, and so no more navigation is necessary. Instead, we'll just read down the page to find the row that we're looking for. It's the second row on this page, and the row after it has an amount larger than 125.86, and so we're done. There cannot possibly be more matching rows as the rows in the index leaf level are sorted by the index key values. With a seek operation, the value or set of values that we search down the index tree for is known as a seek predicate. Let's contrast this with a scan. This time, we're summing all of the amounts. There's no value that we're looking for and so the index tree is not helpful here. Instead, we're just going to read all the rows in the entire leaf level. A scan doesn't always read all of the rows in the table. In this case, it has, but it doesn't have to. What makes this a scan is the fact that it's not searching down the tree for anything, but it's reading the leaf pages directly. We can also have a scan when we're searching for a value, but can't navigate down

the index tree to find this value. In this case, I want to sum the amounts, but only the ones over 250. I can't seek. The amounts over 250 are scattered all through this index, and so the index tree is not very useful here. This again is going to be a scan, but this time it's a scan with a predicate applied. Every row in the table is still being read, but this time, after it's read, the predicate is applied, and the row is only returned if the predicate returns true for that row. The difference between this predicate and a seek predicate is that for a seek predicate, unwanted rows are never even read. For the predicate as seen here, the rows are read and then discarded if they don't match.

How Index Operations Appear in Execution Plans

In this demo, I want to have a look at how seeks and scans show up in SQL Server's execution plans and examine some of the more important properties of those execution plan operators. For this demo, I'm going to use my transactions table. If we expand out the table in Object Explorer and expand out the Indexes folder, we can see that it currently has only one index, the primary key on TransactionID. Let's create another index on TransactionType and Amount. And now if we go back to Object Explorer and refresh it, we can see that new index. And we can look at its properties and we can see the two columns that it's on. Alright, so first up, let's have a look at how SELECT shows up in the execution plan. We've got a single index seek operator. The SELECT is just a representation of the query itself. And if we hover over the operator, we get the tooltip with a whole lot of information in it. The interesting things here to note are the Seek Predicate, which shows us that we did seek on both TransactionType and Amount. We can see the number of rows read, which was 1, and we can see the number of rows returned, actual number of rows, which is also 1. So this tells us that this was very efficient. This query read a single row to return a single row. That's about as good as you can get. Right-click on the Operator and select Properties or press F4 with the Operator highlighted, and we get the Properties window. Here we can see the same information, just in more detail. And if we go to our Messages tab, I turned on statistics _____ while I ran this query, and we can see that this query did two logical reads. That would be the root page of the index and the leaf page of the index. That second one I want to have a look at is a scan. This is going to be a full scan of the table because there's no WHERE clause, and so no rows are going to be filtered out. Execution plan's a little more complicated here. Most of that's because of the sum. The Stream Aggregate and the Compute Scaler are just there to calculate the sum. What we're interested in here is the index scan. We can see the object, which gives us a

table name and index name, much as we had in the previous one. This time though, there's no seek predicate. There wouldn't be. This is a scan. And we can see that we read 26, 268 rows, and we returned from this operator 26, 268 rows. Again, we can pull the Properties window up if we want to have a look at any of those in more detail. Alright, lastly, a scan with a predicate. Again, we've got a sum. This time we're filtering for an amount greater than 2, 500. This will be a scan. There's no index that can support this for a seek operation, and so we're going to go scan with a predicate. The execution plan superficially looks the same as for the previous query when we had to scan without any predicates, and that's because the details of the predicates will be hidden inside the index scan operator. They don't show up separately. If we hover over the index scan operator, we can see that we now have a predicate, which is on the amount of which is filtering for greater than a particular value. Note the difference between this and a seek predicate. You can get both of them in the tooltip when you've got an index seek that applies some seek predicate, but then has to apply predicate afterwards. This being an index scan, just has a predicate. The one thing to note in this one is that for the first time, our number of rows read and actual number of rows are different. 26, 268 rows were read from the table, but only 13, 223 rows were returned to the next operator, the stream aggregate. We threw away over 13, 000 rows because they didn't match the predicate. We read them and then discarded them. This is not ideal if you're trying to tune a query because that's wasted work. And if I go and look at the Messages tab here, this time we read 54 pages. That'll be the entire leaf level of the index. I didn't show the messages in the second query, but if you try that out, you'll see that that one also had 54 logical reads because again, it was a scan that read the entire leaf level. In this module, we've looked at the basics of index architecture, and we've seen how those indexes are used by SQL Server. We also looked over SQL Server execution plans and saw how each of the index operations show up and what some of the more important properties are. In the next module, we'll dive into the details of clustered indexes, a special type of index in SQL Server.

Designing Indexes to Organize Tables

What Is a Clustered Index?

Hi, and welcome to this module on Designing Indexes to Organize Tables. In this module, we're going to look at one specific type of index that SQL Server has, the clustered index. This index has a profound impact on how the table is stored and organized and hence, deserves special consideration. We're going to start by looking at what clustered indexes are and why they're important. We'll spend some time on a common and persistent debate that exists on how the clustered index is used and examine both sides. We'll look at the guidelines for clustered index, and then I'll finish this module with my personal approach to deciding on clustered indexes for tables. What then is a clustered index? In the previous module, we saw that SQL Server's indexes are balanced trees, and I described the bottom level of the index as the leaf level. A clustered index is one where the pages in the leaf level of the index are the actual data pages of the table. The data pages, which comprise the leaf level, contain all of the columns which the table has, borrowing some of the largest data types. This means that a clustered index enforces the logical order in which the rows of the table are sorted, and it defines how the table is stored. Because the clustered index has all of the table's columns in its leaf level, it will be the largest index for that table, and it will be affected by all data modifications made to that table. Since the clustered index is an index built on the actual data pages of the table, there can only be a single clustered index on a table, or none. It's entirely allowed to have a table that doesn't have a clustered index. Such table is called a heap. We could also have a clustered columnstore index, and that I'll discuss in a later module. Since there's only one clustered index on a table and since it defines the table storage, it does need to be chosen carefully. There are two schools of thought as to how a cluster index should be chosen. The first says that the clustered index is important to organize the table and should be chosen predominately with that in mind. The second says to rather use the clustered index to support the most common access path, that is the most common way to filter the table and not worry how the table is organized. I generally go with the first, and that's what this module focuses on, and I'll discuss the tradeoffs in the demo and later in this module. For a second, the design considerations for the clustered index will be much the same as for the non-clustered indexes, which I'll discuss in the next module.

Clustered Index Guidelines Narrow

The guidelines for a clustered index are usually given as four keywords. Ideally, the index key should be narrow, unique, unchanging, and ever-increasing. These are guidelines, not hard and fast rules, and there will be times when one or more should be ignored. So let me have a look at them in detail. Narrow refers to the size and bytes of the index key column. The larger the index key is, the more overhead there will be in the index. For clustered indexes, that's mostly about the size of the rows on the intermediate pages and about how many levels of intermediate pages there will be. The lowest practical size for clustered index key is around 4 bytes, and the maximum allowable size for clustered index key is 900 bytes. The key size doesn't affect the leaf level. As for a clustered index, they will always contain all of the columns in the table and hence, the number of leaf pages is a factor of the size of the table, not the size of the index key. For intermediate pages, however, only the clustered index key is present and hence, the larger the key, the more intermediate pages are required and hence, more levels are required. Let's have a look at how that'll work. Let's say that we have a table with a million rows with the leaf pages containing 100 rows each. For that, we need 10, 000 leaf pages and hence, 10, 000 rows need to fit into the next level above the leaf. At 4 bytes per key, 40, 000 bytes is required, and that fits into 5 pages. That makes this an intermediate level. We then need a single root page above that. If we look at the other extreme, a 900-byte clustered index key, the number of pages that we need in the first intermediate level is 1, 125. Given that, the next level of intermediate pages will have 127 pages in it. We still can't fit 1 row for each of those 127 pages into 8 KB, and so we need a third intermediate level with 15 pages. Fifteen rows need to go into the next level up. And at 900 bytes per row, that comes out at 13.5 KB, which means we need still one more intermediate level with two pages and then finally, a root page. Our index with a 900-byte key is 6 levels deep, and that's with only a million rows in the table. There's another problem with wide clustered index keys, and that's the overhead that they add to other indexes. The clustered index key is used as an address for the rows, meaning it's added to every single other index on that table. When the clustered index key is large, that overhead adds up very fast. So that was narrow.

Clustered Index Guidelines Unique and Unchanging

The second guideline is unique. The clustered index has to be unique. It's used as the row's address. If it's not unique, that is, if it's not created with a unique or primary key keywords, then SQL Server assumes that it is not unique. If that is the case, an additional column will be added to the clustered

index key, a hidden column that goes by the terrible name of the uniquifier. This column is a 4-byte signed integer and it's null for the first occurrence of any value or set of values in the clustered index key and then given a value starting at 1 for any subsequent appearances of the same value or set of values. To be honest, this is the guideline that I break most of the time. I'm not worried about putting the clustered index onto a column that's mostly unique. Now I'm still not going to put the clustered index onto a column with only a couple of distinct values in millions of rows, but on a reasonably unique set of values, it's fine. The worst you're going to get by ignoring this guideline is an extra 4 bytes on your index key. Oh, and yes, it is possible to get an error from running out of uniquifiers. Considering that that requires 2.1 billion duplicates of the same value for the clustered index key, this shouldn't be something you're worried about. On to unchanging. The index key defines where in the index the row belongs. If we image a scenario where the clustered index in on an integer, then the row with a value of 200 for that key column belongs on the page that has values between 170 and 250. If the row is updated and the clustered index key column changed to say, 70, that row has to move. This change is the first update operation into a split, delete, and insert. The row is deleted from the table, removed from the clustered index and any other indexes we may have, and then reinserted into the new location and added to any other indexes. In an OLTP-type environment, this can have noticeable effect on the throughput of the system, hence, the recommendation is that the clustered index key column be one that doesn't get changed after the row has been created.

Clustered Index Guidelines Ever Increasing

Finally, ever-increasing. This one can be a little hard to see the reason for. The underlying reason for this recommendation has to do with some specific internal behavior of the SQL Server database engine, specifically, how it handles a scenario where a row has to go onto a particular page, but the page in question is full. What happens in that case is a page split, and on the next slide, I'll show you exactly what a page split looks like. Frequent page splits effect inserts mostly, although, they can also affect updates, and it slows them down. As with the previous guideline or a busy OLTP system, this can have a noticeable effect on the system's throughput. Having the clustered index key column always be increasing, meaning each row inserted has a higher value than anything existing in the table, means that mid-index page splits don't happen. We're always inserting into the last page of the index. This sounds like it might cause a hotspot problem, and in some circumstances it can, but that

usually occurs a very high insert to second, far above what most OLTP systems will experience. The usual solutions for insert hotspots in recent versions of SQL Server involves in- memory tables, not a different clustered index key. This slide's a little technical. So feel free to skip the rest of this clip if you're not interested in the exact mechanisms behind page splits. So we have our clustered index here, and our clustered index key is transaction type and amount. All of the pages in the leaf level of this index are full. Let's assume we can only fit five rows onto a single page. We need to insert the row C 200.45. This has to go onto the page that starts with C 65.23 and ends with C 264.36 because the index key defines the logical storage order. That page, however, is full. What SQL Server does in this situation is to allocate a new page with half of the rows on the exiting page onto that new page, and then make the necessary changes to the intermediate page and adjacent pages and insert the new row. That's a lot of work for a single-row insert, and this is why frequent page splits can cause problems on OLTP systems. I want to show that new page at the end of the index, but logically it's not. Logically, the order of those last 5 pages is 1, 2, 3, our new page, and then back to 4, and 5.

Demo: Effects of Different Clustered Indexes

In this demo, I'm going to create various different clustered indexes on a table and show you how the choice of key affects the table size, the speed of inserts into that table, and the speed of updates to that table. For this demo, I'm using three copies of my transactions table, which have been padded out with extra data, and I'm going to create three different clustered indexes on those tables. One, a very wide clustered index, one on Amount, which is neither unchanging nor ever-increasing, and one on TransactionDate, which is reasonably narrow, mostly increasing and unchanging. And yes, I realize the amount probably wouldn't change in a real system, but let's pretend that it does in this one. So I'm going to create those three tables and those three clustered indexes, and this will take a while, so I'll just pause the demo now and come back once it's all finished. And we're all done. Each of those tables has approximately 27, 000, 000 rows in it. The first thing I want to do is look at the relative sizes of the three tables. I'm going to do this with a DMV called sys.dm_db_partition_stats that will give us, among other things, the number of pages allocated to that table, or more specifically, that index. And these three tables only have a clustered index. And if we look at the results, the table with the very wide clustered index is the largest as one would generally expect, followed by the table clustered on TransactionDate and finally the one clustered on Amount. The difference is not particularly large in this

case, but that's because these are very small rows. There aren't a lot of columns in this table, and so we don't have a very dramatic effect from the size of the clustered index. The TransactionDate column is indeed wider than the amount. TransactionDate is a 13-byte data, while the amount, I think, is an 8-byte numeric, and that's why the TransactionDate table is slightly larger than the table clustered on Amount. For the second test, I've got two store procedures here, one which inserts a row and one which updates a random row. And I've got a little C# application, which I wrote, which we'll call one or the other of these store procedures in a parallel loop 1, 000 times. This is just to test, update, and insert speeds with the different clustered indexes. So let me start with updates, and we'll change the UpdateRows store procedures so that we're updating the table that has a clustered index on TransactionDate. And we're updating our 1, 000 rows in anything up to about 100 milliseconds. This is pretty decent even on a desktop PC. Let me change that so that we're updating our table with a clustered index on amount. Since we're updating the amount column, that means that our updates and our split pair of delete and insert, and we want to see if this is going to be slower, and if so, by how much? Fourteen seconds, ten seconds, nine-and-a-half seconds. This is significantly slower. Now admittedly, I'm not running this on a top-end server, but still, this is a fairly significant difference just from a different clustered index. That was our updates. Let's see how badly the clustered index choice affects inserts. I'm going to change my application quickly to call the InsertRow procedure. I'm going to run that first against the table cluster by TransactionDate. Looks like anything from about 70 to 180 milliseconds, again, pretty decently quick to insert 1, 000 rows in parallel in that kind of time. Let me change that so they insert into the other table, the one that's clustered on Amount. Two-and-a-half seconds, four-and-a-half seconds, two-seconds, one-and-a-half seconds. This is significantly slower and that's because we're getting page splits from our inserts because they're inserting into random pages. So that's a quick look at how the choice of clustered index affects your insert and update speeds, as well as the size of your table.

Clustered Index Guidelines Trade Offs and Compromise

Now that we've look at how the choice of clustered index affects size, inserts, and updates, let's layout the tradeoffs between choosing a clustered index key to organize the table and choosing a clustered index key to support queries. If we choose a clustered index to organize the table, the table will be as small as it can be. We'll have minimal page splits and we'll have less overhead on updates. We will,

however, need additional non- clustered indexes to support our workload. If we choose to support queries and not worry about the table's organization, then the table may take up more space than necessary. How much more space depends on exactly how wide the clustered index key is. There will likely be problematic page splits from inserts and updates, and there might be additional overhead on updates from splitting them into deletes and inserts, might. It depends on the type of system whether those will be a problem or not. For a high-load OLTP system, they're quite likely to be problematic, but for a data warehouse-type environment, the overhead on inserts and updates might be completely consequential. And finally, of course, if a clustered index supports queries, there will be less of a need for additional indexes, not no need at all. It's highly uncommon to find a table's only ever accessed by one set of columns, but less of a need. And when I discussed earlier these two schools of thought, they're not two completely distinct scenarios with no ground between them. Quite often, it's useful to take a position between the two when choosing clustered index keys. Indeed, in many cases when I'm tuning systems, I look for possibilities where I can choose a clustered index key. There's a compromise between the two extremes, a clustered index key that's fairly narrow, unchanging, and ever-increasing. A common type that's often a good choice for this is a date in any kind of transactional table. Dates are usually fairly narrow, coming in at 8 to 12 bytes in most cases. And if we're dealing with a transaction date, then it should be unchanging and mostly always getting larger. And those kinds of tables often query by date. So we get a fairly narrow clustered index key that's hopefully unchanging, mostly ever-increasing, and it supports one of the more common query Xs passed to the table, and so we've got the best of both worlds, not one or other narrow extreme. Now this isn't the only possible column type. It's just one example that I find frequently in client systems. I encourage you to have a look at your tables and choose your clustered index keys with an eye on the tradeoffs, decide which tradeoffs you can live with, and then test to make sure that your decision is acceptable. In this module, we've looked at what clustered indexes are and how they're architected. And then we spent a lot of time on the clustered index debate and guidelines looking at the desirable properties for the clustered index and the tradeoffs for choosing clustered index keys with or without those properties. In the next module, I'll discus non- clustered indexes, which are the index type most commonly used to support queries. See you there.

Designing Indexes to Improve Query Performance: Part 1

Introducing Nonclustered Indexes

Hi, and welcome to this module on Designing Indexes to Improve Query Performance. In the previous module, we looked at design strategies for the clustered index, mostly focused around using the clustered index to organize the table. In this module, we're going to focus on a second type of index that SQL Server has, the nonclustered index. We'll start by looking at what the nonclustered index is and how it differs from the clustered index. We'll touch briefly on common query predicates, not in detail, but just enough so that I can talk about how to design indexes for each one. And then we'll go into indexing in detail, lots of detail. We'll look at how to create indexes for queries that filter with various different types of predicates and also for joins. We'll then look at include columns, and what benefits they bring, and the pros and cons to using them. And finally, we'll talk about filtered indexes and where they're useful. First though, what exactly is a nonclustered index? Well, let's say that I'd finally written that book on indexing that I've been considering for years, and it's nearly finished. My editor mentions that he still needs the index for the back of the book. A part of that index might look like this. And if you have a technical book nearby, you can grab it and open it towards the back. There's a good chance that book will have an index. The index in a technical book is a separate section to the contents of the book itself, containing a list of keywords and what pages they may be found on. If I wanted to use the index in this book to find about nonclustered indexes, I wouldn't have to read the whole book to find the information. Instead, I go to the index, and I see that indexes, nonclustered, is found on page 45. I then go to page 45 to find my information. A nonclustered index in SQL Server is quite similar. It's a separate structure from the table, unlike the clustered index that we looked at in the previous module, and they're used to locate rows in the table. Because they're separate structures, it's possible to have more than one of them on a table. The current maximum is 999. Please note that's a maximum, not a goal. The most I've ever seen on a single table was 76, and that was far more than it needed. It's generally far more normal to need maybe 10 nonclustered

indexes to support the entire query workload of a table. Because they're separate structures from the table, they don't have to, and in most cases, shouldn't contain all of the columns the table has. A nonclustered index should be created on the minimum number of columns needed for what it was designed for. This is in contrast to the clustered index, which always contains all of the columns in the table because it essentially is the table. Another implication for nonclustered indexes being separate structures is that they are partial duplicates of the data in the table, and as such, are storage overhead. This isn't usually a massive concern, but it can make a database larger than needs to be if you've got a lot of nonclustered indexes on larger tables. The nonclustered index is always in sync with a table. It always has the same number of rows in it as the table does and any data changes made to the table are immediately made to the nonclustered indexes as well. This has implications for data modifications when there are lots of indexes, and it's something I'll discuss later. Before we get into the details of creating indexes to support the various queries that run against your tables, I'd like to go over the common types of predicates that queries have. Equality predicates are the simplest, direct comparison of a column to a constant variable or parameter. For indexing purposes, we're going to look at single equality predicates and multiple equality predicates combined with an AND. The second type are inequalities. I'm only going to explicitly cover inequalities that are unbounded on one side, that is greater than or less than, but exactly the same indexing principles will apply to betweens or any similar combination of greater thans or less thans. I'll look at how to index a combination of equality and inequality predicates and how to best handle multiple inequality predicates on different columns. We'll cover how combining predicates with ORs complicates indexing and how to best to handle it. And finally, we'll cover when and how to create indexes and support joins.

Equality Predicates

Starting with equality predicates. For a query to be able to use an index for a seek operation, which is usually what we want, the query must filter on a left-based subset of the index key. This sounds complex, but it's nothing more than saying that if we have an index on 3 columns, say column 1, column 2, and column 3, a query can only seek against that index if it filters on column 1, column 1 and column 2, or all 3 columns. The second of these guidelines is that when considering a single query with two or more equality predicates, SQL Server can use with equal ease indexes with the columns in different orders. So for a query that has equality predicates on column 1 and column 2, an

index on column 1, column 2 is just as useful and usable for seek operations as an index on column 2, column 1. Now this isn't to say that index column order is irrelevant. Quite the contrary. The order of columns in the index key is crucially important when designing indexes that multiple queries can use, and this is our goal. We don't want to craft an index specifically for a single query, but we rather want to create indexes that as many queries as possible can use as efficiently as possible. Let's have a look at those guidelines and the index architecture and see exactly why they work this way. Importance of filtering on a left-based subset of the index key. Let's say I have an index on two columns of my transactions table. The first column in the index is Amount and the second is TransactionType. I have a query that filters on Amount. It's looking for any rows where the amount is 200 exactly. Starting from the root page of the index, I can see that there are two pages below the root. The first has the lowest value for an amount of 10.25, the second has the lowest value for an amount of 264.36. The rows we're looking for cannot be beneath the second of those. We're looking for a value of 200, and the second branch starts higher than that, hence, we follow the link to the first page. This is an intermediate page, not a leaf, hence, there's still one more level of pages beneath us. And so we repeat the process of identifying which link to follow. The four pages beneath this one have lowest values for the amount of 10.25, 75.12, 157.56, and 207.56. The rows we're looking for are not going to be on the first or second pages, they're too low, and it's not going to be on the fourth because it's too high. The third page, however, covers the range between 157.56 and 207.56 and hence, any rows with an amount of 200 will be on that page, so we follow that link. This is a leaf level. There's no more levels beneath us, so we can simply read down the page until we find the rows we're looking for. In this case, there are none, and once we find a row with an amount above 200, we can conclude that, and we can abort the seek returning 0 rows. Since the index is ordered by its key column, once we find a row with a value above the one we're looking for, there cannot possibly be more rows with the value 200. That was simple enough. How about this though? This time, we're filtering for TransactionType of D. So we go to the root page and then what? I can't tell where the rows with the TransactionType of D are. The index is ordered first by Amount and after that, by TransactionType, and hence, the rows with TransactionType of D are scattered across the entire index. I can't do a seek here. There's no way that I can start at the root page and search for the values I want. The only way to find the rows with TransactionType of D is to scan this index, to read every single leaf page, and filter out the rows I don't want, hence, the importance of creating indexes for queries with index keys such

that the queries filter on a left-based subset of the index key. Only then can they perform a seek operation on that index. Anything else has to be scan. Let's have a look at the second guideline. When considering a single query with multiple equality predicates, the order that the filtered columns appear in the index don't matter. Again, we have an index on Amount and TransactionType, and this time, I have a query that filters on both of them. As with the previous example, we can navigate down the index tree following the first link because 200 D is less than 264.36 B, then the third link because 200 D is between 157.56 D and 207.56 C, and we can find on the leaf page that there were no matching rows. If the index columns had been reversed, I could follow very much the same process. From the root page, in this case, I want to follow the second link. D 200 is larger than B 297.45 and hence, the row I want will be found by following the second link. From the intermediate page, we can see that the last link is the one that needs to be followed. D 200 is larger than D 157.56 and hence, the only place the row could possibly be found is on that last page. Note we're seeking on both columns. We're not seeking on the first and then filtering out the second. We're seeking directly on both columns. That's why I can ignore the third link from the intermediate page. It may contain some transaction types with D, but it can't possibly contain D 200 because the lowest value for the fourth page is D 157.56. Following the link down to the fourth page, I can again see there are no matching rows. As with the other index, once we find a value larger than the row we want, in this case, D 248.32, we can stop. The index is ordered by its key columns. There can be no more matching rows. We did exactly the same amount of work in these two cases. We read three pages, we followed two links, and we found no rows. Hence, for this single query, the order that the index keys were in was not relevant. That was one query, however. What if we have two queries that we want to use the same index? Let's say that we've now got a second query where TransactionType equals B. Our first query we know can seek on this index, Amount of 200, TransactionType of D. This is exactly the same of an earlier example. We seeked on the first link, then the third, and we find there are no matching rows in that table. What about the second one though? The second query can't seek. TransactionType is not a left-based subset of this index. The index's leading column is Amount. Therefore, our second query filtering only on TransactionType has to scan the index. This is far from ideal. Let's switch the index column order around and try those two queries again. This index is now TransactionType and Amount. The first query we know can seek. We've seen this before. Follow the link to the second of the intermediate pages, then to the last of the leaf pages beneath it, and read down to find that there is still no row for

D200. What about the second query? Well, this time it can seek. It's filtering on a left-based subset of the index key. The index is a TransactionType and Amount and this query filters on transaction type, and so it can seek, and it seeks to the first of the intermediate pages because that's where the TransactionTypes of B start. From there, we want to follow the third link. That page starts with A, but it may have some rows with TransactionType of B on it because the next page starts with B, and indeed it does. And we read down the page to find the first row that has a TransactionType of B, and then we read along the leaf level until we find the first row that has a transaction type greater than B, at which point we're done. We need to do no more reads. We've read three pages in the leaf level, five pages in total. This is the ideal. Multiple queries all being able to effectively use a single index. When we're indexing systems for good performance, this is what we want. As many queries as possible using as few indexes as possible.

Inequality Predicates

Let's have a look at inequality predicates, fairly similar and slightly more complicated. The first guideline we've seen before. It's the same as for equality predicates. To be able to seek against an index, the query must filter on a left-based subset of the index key. This is for exactly the same reasons as for equality predicates. The second guideline is that when a query has both equality and inequality predicates in it, for it to use an index as efficiently as possible, the columns that are filtered on by the equality predicate should go earlier in the index than the columns filtered on with the inequality predicate. Third, less a guideline and more of a caution, when a query has multiple inequality predicates on different columns, it's very hard to index that well. It's possible to create good indexes for it, but it is tricky. Let's go back to our index tree and see why those guidelines exist. The first one we're already familiar with, left-based subset of the index key. I'm not going to spend a lot of time on this. Again, here I have an index on Amount and TransactionType and I have a query filtered for Amount greater than 200. 200 is greater than 10.25, it's less than 264.36, so we follow the first link, then we follow the third, then we go down the page. This time we're not looking for a match, we're just looking for the start of a range. So we start reading from the first row that has an amount greater than 200, and we read to the end of the index because this is an unbounded inequality, You'll sometimes see this referred to as a range scan. I personally find that term somewhat misleading. This is a seek operation. It's not a scan. It's navigating down the tree looking for a value. It will show up in execution

plans as a seek operator. And same as we saw for equality predicates, if I have a query filtering on TransactionType alone, it can't seek on this index. It has to do a full scan. Next, the order of columns in the index when they're both equality and inequality predicates. We've got our index on Amount and TransactionType again. This time, the query is looking for rows where the amount is greater than 200 and the TransactionType is D. I'm sure this is all starting to feel very familiar. We seek to the start of the range where Amount is equal to 200 and then we read the rest of the index; however, we're not returning all of those rows. Every one of those rows gets a secondary predicate applied to it, TransactionType = D, and we throw away all the rows that don't have a TransactionType of D, which is the majority of them. We've read in total eight pages to get this data, root, intermediate, and six leaf pages. What if I flipped the order of the index columns around? The index is now TransactionType and Amount. My query's unchanged. D 200 is greater than B 297.45, and we take the second link. We want the fourth link as we've seen before when we're looking for TransactionTypes of D, and now we're looking for TransactionTypes of D where the amount is greater than 200, and we find there are only 4 of them. So we read down the page to find the start of the range, and then read the rest of the pages in the leaf level, which is none of them because we're at the end of the index already. We've read three pages this time, root, intermediate, and a single leaf page, and we haven't thrown any rows away. Every row we've read, we've returned. This is why when designing indexes for queries with equality and inequality predicates, you generally want to put the columns for the equality predicates first and then the inequalities so that your seek is more efficient and so that you read only the rows you actually want to return and don't read a large number of rows, apply a secondary predicate, and throw a bunch of those rows away. Now this isn't always possible, especially when you're creating indexes to support multiple queries, but it is something you want to try for when tuning important queries and when looking at your workload overall. Lastly, I look at why indexing queries with multiple inequalities is hard. Back to our index on TransactionType and Amount, and this time my query has two inequalities. This query has an unbounded inequality on TransactionType, and that's my leading column, so what we're going to need to do here is find the end of the range. What we're doing here is looking for the first row with a TransactionType that is greater than B, and so we read to the second intermediate page, down the first link this time to find the first row with a TransactionType greater than B, and then we read backwards down the index. Again here, we're reading rows that we don't want, applying a secondary predicate, and throwing them away. In this case, we're throwing away any rows

with an amount less than or equal to 200. We've read seven pages for this query. What about the index in the other order? This time this doesn't help. With the index leading column B in Amount, we take the left link because that starting value 200 is less than 264.36. We take the third link, we read down the page to find the first row with an amount greater than 200, and then we read to the end of the index. And again, we're going to be throwing away rows we've read. This time, rows with a TransactionType greater than B. This time, we've read eight pages. Not a massive amount more inefficient, but in both cases, we're reading data that we don't want, and with multiple inequalities, there's actually no way to get around that. When you're dealing with multiple inequality predicates, only one of them can be used as a seek predicate. Any others have to be applied to secondary predicates, filtering out rows once they're read. With multiple inequalities, it can be very difficult to decide which column order is better. What I recommend in most cases is testing out the various options on representative data and representative queries and going with whichever has the better performance for the queries and values that your application typically uses.

Demo: Equality Predicates

So far in this module, we've spent a lot of time manually navigating down index trees to see how they'll be used. Now I want to try a variety of queries against a real table with indexes to see how they behave. I've got three queries here all against my transactions table, one filtering on ClientID, one filtering on TransactionType, and one filtering on both. Currently, the table has no indexes other than its primary key. And if I run the first query and look at the execution plan, I can see that it's doing a clustered index scan, essentially a table scan. There's no index that it can use. And if we go to the Messages tab, we can see that this query did 166 logical reads. That will be the size of the entire table. It's filtered on a single column, so there's no question of what our index column order should be. Let me create an index on ClientID. And now when I run that query again, you can see that it's using an index seek instead. We're not accessing the clustered index any longer because the index we just created supports this query. That index though doesn't have TransactionType in, and if I run the query on TransactionType, it's still doing a table scan. It has to. There's no valid index here. So again, I can create an index on TransactionType to support this query. And if we run that query now, it's doing an index seek on our newly created index. The third query filters on both columns. I don't have an index with both columns in. I've got two single-column indexes. Now this is something I see all too often in

reality. I often see systems where there are lots of single-column indexes on the tables. Unfortunately, they're not generally very useful. If I run this query, you can see that SQL actually has used both indexes, but what it's had to do is seek on both indexes and then do a merge join to do an intersection, to find the rows that match both the ClientID and the TransactionID. This is called an index intersection. You won't see this all too often. It is sometimes chosen as it can be more efficient than a table scan, but it really depends on the table, the number of columns, and the amount of data that you're fetching from the table. It's definitely not ideal. It's not the most optimal we could possibly be. So let me drop that index on TransactionType and create an index on TransactionType and ClientID. Which index I dropped to create the wider index is really a matter of choice here. I could have dropped the index on ClientID and created a new one with ClientID TransactionType. That option would have worked just as well. And now if I run the query that filters on ClientID and TransactionType, I can see that we've got a single index seek operation. If I rerun the query that filters on TransactionType alone, it's also using this new index on TransactionType and ClientID. But if I go and drop the index on ClientID, then my query that filters on ClientID is not going to be able to use that index, at least not for a seek operation. And we can indeed see that it does an indexes scan on that index on TransactionType and ClientID because ClientID here is not the left-most column of the index key, and so this query can't seek.

Demo: Inequality Predicates

As in the previous demo, I've got three queries here against my transactions table, one with an inequality on Amount, one with inequality on ClientID and an inequality on Amount, and the third with an inequality on Amount and another inequality on TransactionDate. I dropped all of the indexes that I created in the previous demo, so again, there's only the clustered index on this table. So the query that filters on Amount is doing a clustered index scan because there are no other indexes on this table. So let me create an index on the Amount column and run that query again. And this time, you can see that we've got a seek operation, and we're reading eight pages from this table. For the second query, we're going to want to index on Amount and ClientID. Let me start by creating an index on Amount and ClientID, in that order, and see how that one works. And if I run the query, you can see that we've got an index seek. So it looks like this query is efficiently using that index, but it's not using it as efficiently as it could be. If we look that I/O statistics, we've got nine logical reads. Let me create another

nonclustered index with the columns in the other order, ClientID first, then Amount. If I run my query again, you can see that this time, SQL Server has chosen to use the index on ClientID and Amount, not the index that it previously used, which was the one on Amount and ClientID. And if we go to the Messages tab, we've only read two pages to execute this query. We've dropped from nine to two, which in terms of numbers, is not overly impressive, but this isn't a very large table. In larger tables, I've seen reductions from tens of thousands to about 20. Alright, so let me drop the index that I created earlier on Amount and create one on Amount and TransactionDate for our third query. And our third query uses that index quite happily, but if you look at the tooltip, we've got a Seek Predicate on Amount and then a Predicate on TransactionDate. We were able to use the index to seek for the Amount, but we weren't able to use it to seek for the TransactionDate, and so that's got to be executed as a secondary predicate, and that query with that index did 10 reads. So let me drop that one and create an index with the columns in the opposite order, TransactionDate and Amount. Our third query uses that index now, but again, we've got a combination of seek predicates and predicates. This time, we're going to seek predicate on TransactionDate and we've got a predicate on Amount. We can seek for the TransactionDate, but we couldn't seek for the Amount, again, because this is two inequalities, we can't seek for both of them. And if I go and look at the Messages tab, this one's done 15 logical reads. So if this was representative data and representative filter values for my production system, I could conclude that the index on Amount TransactionDate is more efficient than the one on TransactionDate and Amount. Although, to be honest, quite often it needs a lot more testing and it's nowhere near as clear cut as that.

Indexing for Predicates with ORs

Moving on to some slightly more complex indexing scenarios. First, how optimal indexes differ when working with predicates combines with ORs. In our previous examples with multiple predicates, we had the predicates combined with ANDs. ANDs are easy. Each additional predicate reduces the result set, or at minimum, doesn't affect it. And so, any number of predicates combined with ANDs can be executed by finding rows that satisfy one or more predicates via an index seek and then applying the other filters as necessary. ORs are the opposite. Each subsequent predicate increases the result set, or at minimum, doesn't affect it. A bunch of predicates combined with ORs requires that each predicate be evaluated separately. The matching row is located and then the results combined. As such, when a

query has multiple OR predicates on different columns, for base performance, multiple indexes are necessary. Let's go back to our index tree and see why this is the case. This time we're looking for rows with amounts greater than 200 or with a TransactionType of B. The rows with amounts greater than 200 are all adjacent. They have to be. The index is sorted on amounts since it's the leading column. The row is with a TransactionType of B, or not. To satisfy this query, we need to find all the rows that have an amount greater than 200, all the rows with a TransactionType of B, and combine those two sets of rows to get the final set. We know how to get rows with an amount greater than 200 by this point. It's a left-based subset of the index, and so we can simply navigate down the index, find the start of the range, and read the rest of the rows. That's half the query. To find the rows with a TransactionType of B though, we can't seek this index. TransactionType is not a left-based subset of the index key, and so we'll need to scan. And how this will play out in reality is that SQL Server won't bother doing a seek at all. It will do a single scan of the index and apply the two predicates to the results. To be able to seek optimally, what we need is two indexes, one on Amount and one on TransactionType. That way, the rows matching the amount can be found by seeking on the first index, the rows matching the TransactionType can be found by seeking on the second, and then the results can be combined and returned to the user.

Demo: Predicates with ORs

I'm going to demo some queries with some more predicates combined with ORs and then some more complicated mixtures of ANDs and ORs, and we'll see what indexes we need to get optimal seeks. This time, I've got two queries, one with a fairly simple WHERE clause, two predicates combined with an OR and then an AND that applies to everything. And so we'll look at what indexes will work with these. First, I'm going to try an index that has both ReferenceShipmentID and InvoiceNumber in it. The query uses that index, but it uses it for a scan, not for a seek because while it could use that to seek for ReferenceShipmentID, the predicate for InvoiceNumber would require a full scan, and so we only get a single full scan. And we can see that the entire WHERE clause, both columns, are evaluated as a predicate. We've got no seek predicates happening here. So let me drop that and then create an index on each of those columns separately. And if I run my query again, you can now see that we've got two seek operations. The Merge Join and Stream Aggregate are how the OR itself is implemented. The results from both indexes are combined and then duplicates must be eliminated because if a row

has both a ReferenceShipmentID of 452 and that particular InvoiceNumber, we don't want it to appear twice, just once, and so we have to remove rows that were returned by both indexes. In this case, that's done by Stream Aggregate. In more complicated queries or queries against larger tables, you'll see different combinations of operators. The important part here though are the index seeks, one on each of those indexes. And if we look at each of them, you can see that there's a seek predicate for one column, one seek predicate on ReferenceShipmentID and then the other index has a seek predicate on InvoiceNumber. Alright, let's have a look at this more complicated one. It's not immediately obvious how to index this, so we need to fall back on a bit of our Boolean theory and rearrange this WHERE clause. Note you don't actually have to do this in reality. I'm just doing this to show you how this can be indexed. You see, I can move the ClientID = 2875 into the brackets, and then I've got a familiar pattern, two sets of predicates combined with an OR. In this case, I'm going to need one index on ClientID and Amount and another index on ClientID and TransactionType if I want this query to use its indexes efficiently. Since the first combination is an equality/inequality pair, I'm going to want ClientID first in the index and then Amount. And if we run these two queries just to test that they are the same, we can see that they both returned three rows, and if we look at the execution plans, they're identical. SQL's actually gone and logically recombined the predicates in the version of the query that we expanded. Alright, so for this, I need two indexes. Since the ClientID Amount filter is an equality/inequality pair, I'm going to want ClientID first in that index and then Amount as we saw earlier when we looked at inequalities. For the second one, I could do ClientID TransactionType, but since I'm probably going to want this index to support other queries, I'm going to flip the order around and create it on TransactionType and ClientID. Since they're both equalities, the order doesn't matter for this query, it might matter for others. Now if I run this query, and yes, I have changed the value the ClientID filters for to get a few more rows, get a slightly more interesting execution plan, you'll see that we have two index seeks as we expect, one on the index on ClientID and Amount and one on the index on TransactionType and ClientID. This time, we've got a concatenation and a sort that are removing any rows which might have been returned by both index seeks. We're going to take a break here, so get some coffee, tea, or beverage of your choice, and come and join me in the next module where we look at the rest of nonclustered indexing starting with indexing for joins, include columns, and filtered indexes and where they're useful. See you there.

Designing Indexes to Improve Query Performance: Part 2

Indexing for Joins

Welcome to Part 2 of the Designing Indexes to Improve Query Performance module. In this module, we're going to look at indexing for joins, see which of the join types that SQL Server has benefited from indexes and how. We're going to look at include columns and see why their edition in SQL Server 2005 was so beneficial to good indexing. And finally, we'll look at filtered indexes, the ability to create indexes on part of a table. Indexing to support joins is something that I do very seldom, to be honest. It's worth understanding though. There are times where it can make a large difference. There are three physical join operators in SQL Server and any of the three can be used to implement any of SQL's joins, inner or outer. Which physical join is used is decided on by the query optimizer based on low counts in the tables among other things. The nested loop join is typically seen on smaller tables. It's optimal when there are a small number of rows being read from the outer table and it can benefit from an index on the join column on the inner table. Which of the tables on the join is the outer and which is the inner though? Unfortunately, that's going to be decided on by the query optimizer as it generates the plan. It's not based on which table is on the left or the right of the join. Indexing support nested loop joins can be a bit of trial and error. What I suggest is create indexes on both tables on the join columns and on the WHERE clause predicates, if applicable, and then test and see which index the query optimizer chooses. Merge joins require that their inputs are sorted in the order of their join columns. As such, a merge join can use an index since indexes are sorted by their key columns. In scenarios like this, I tend to prefer to create indexes to satisfy the WHERE clause predicates and then only consider whether the index can also support the join by providing an order. Finally, hash joins are the join type used for massive tables. They're typically seen when you're joining millions of rows to millions of rows. They don't benefit from indexes at all.

Demo: Indexing for Joins

Let me do a quick demo here and show you the three join types in SQL Server and how they use indexes differently. This time, we've got a single query on two tables. I'm going to use join heads to change which join is in use so we can see the way the indexes affect the various joins. We have an index on the Stations table on the Location column, so the WHERE clause is already supported with an index, and the primary key of the Station's table is StationID, so one half of the JOIN clause already has an index in the form of the primary key. It's the other half that I'm interested in here. And if we run that query and have a look at the execution plan, we've got a hash join, by default, and a complaint that there is a missing index. And the missing index is indeed on OriginStationID, the other half of the join. So let's try and create an index on the Shipments table on the StationID. If I run the query again, we've now got a nested loop join and two Index Seek operations. The Index Seek on Stations is the same one we had in the previous execution plan on Location, but this time we've got an index seek on the Shipments table, and this is indeed on the OriginStationID. The way the nested loop is working is for each row that comes out of the outer table, that is Stations in this case, a seek is run against the inner table looking for rows of matching OriginStationID. So let me use a join here, quickly, to change that to a MERGE JOIN, and please don't do this in production. This is solely so I can demo the different types of joins. It's best to let SQL choose its join type most of the time. If we look at this execution plan, the index on Shipments is indeed in use, but this time it's for an index scan. There's no filtering we can do here. Merge joins don't seek against the inner table. They read all the rows from the inner table, applying any applicable WHERE clause predicates, and then the merge join walks down both results that's joining the rows. They have to be sorted by the join key, and that's what the index is doing in this case. It's not doing any filtering, and that's why we have an index scan. And lastly, let me force this back to a hash join and have another look. We've again got an index scan on the Shipments table on the newly created index, but in this case, that's solely because it's the smallest access path for the rows and columns that we need, in this case, all the rows and just the OriginStationID and ShipmentID, and the hash join doesn't care about any ordering. It doesn't care about any filters. It's going to use a hash table in memory to do this join. And so the index here is only benefitting us by being a relatively small structure and so requiring less I/Os to read.

Include Columns

Up until now, I've been focusing on the index keys and how they work with query predicates. Predicates, however, aren't the only part of queries. There's also the data that the query returns, that is the columns in the select clause. These can also influence index design, but not in the same way as the predicates do. Rather, they may require that the index has include columns added to it. Include columns are extra columns that I add at the leaf level of the index only. They're not present in the intermediate levels. They're not present in the root page. As such, they can't be used for seek operations. There's no way to navigate down the tree and search for value for an include column. What they're useful for is avoiding key lookups. Key lookups are single-row seeks done against the clustered index and only against the clustered index to fetch columns needed by the query that are not present in a nonclustered index. Let's have another look at our very familiar index tree. This time, I've added a SELECT clause to the query. We're selecting the Amount column, the same column we're filtering on, and a column we haven't seen before called Is_Priority. We know how to fetch the rows matching the amount, but there's a problem this time. We want the column, Is_Priority, and it's not in this index. All this index contains is Amount and TransactionType, hence, just like with the index at the back of a book, I need to look the missing data up. That's called a key lookup. I mentioned key lookups briefly in an earlier module. They're single-row seeks against a clustered index, and they're made to fetch columns, which are necessary for the query, but are not present in the nonclustered index used. Since the nonclustered indexes always contain the clustered index key, for each row fetched from the nonclustered index, SQL can perform a seek against the clustered index to get the rest of the columns. Now that's probably fine if there's only a couple of them. A seek against a nonclustered index for a single row will probably read two or three pages. And if a key lookup is necessary, that's probably another two or three pages. But what if there are a lot of rows? What if our seek against a nonclustered index returned 1, 000 rows? That's probably only still a few page reads for the seek, but the key lookups are done one by one. If a single key lookup requires 3 reads, then looking up 1, 000 rows requires 3, 000 reads. Because they're done one row at a time, key lookups are slow for large numbers of rows, and the optimizer usually won't choose to seek on a nonclustered index if it will have to do a lot of key lookups. And by large, I'm talking about half a percent of the rows in the table, so this is why we have include columns. What we can do is include the extra columns needed at the leaf level of the index. And then, once the Index Seek has found the rows, the needed column is right there and no key lookup is necessary. There are downsides to this. Include columns

make the index larger, and hence, the table larger, and hence, the database larger. And used excessively, they can make indexes slower to use. It's usually a good tradeoff; however, providing that there aren't multiple nonclustered indexes that include every single other column in the table. That's a little overkill.

Demo: Include Columns

In this demo, I'm going to have a look at key lookups, the tipping point at which the optimizer will choose to scan the table rather than do all the key lookups and how include columns help. For this demo on include columns, I've only got a single query, but it's got a much larger SELECT clause than we've seen before. I've got two WHERE clause predicates, ClientID and TransactionType, much the same as we've seen in earlier demos, but this time I'm fetching a whole pile more columns from the table. There are no indexes on this table other than the primary key, so if I run this query, we're going to get a clustered index scan. Let's start by creating the obvious index, ClientID and TransactionType. And SQL Server does indeed use it, but this time we've got another operator, the Key Lookup. If we look at the Index Seek, it's much the same as we've seen before, two seek predicates on this index. I will draw your attention here to the Output List. TransactionID, ClientID, and TransactionType, those are the only three columns present in this index, and so they're all returned from this index. But there are a number of other columns that we want in this query that aren't in that list, Amount and TransactionDate, and those columns have been fetched from the clustered index by that Key Lookup operator. This isn't fantastic, and on larger row counts the optimizer will choose not to do this Seek and Key Lookup combination and instead just to scan the clustered index, or sometimes scan another nonclustered index because the cost of the lookups is quite high. So let me drop that index and create another one, and this time I'm going to add an include column. Specifically, I'm going to include Amount, TransactionDate, and TransactionID. And now if we run our query again, we're back to a single Index Seek operation. Now let me show you what happens when we've got too many rows for that Key Lookup to be efficient. I've got a slightly different query now, filtering on TransactionType alone, but fetching the same columns. And I'm going to create an index just on TransactionType. If I run that query, we get an Index Scan. We get an Index Scan on our previously created index, the one on ClientID and TransactionType. We can't seek on that because TransactionType is not the left-based subset of the index key, but the optimizer has figured out that it's faster to scan that index than it is to

scan the table. And it's all right, there are a couple more columns on the table. It hasn't gone for an Index Seek on our newly created index on TransactionType and a set of Key Lookups because there are too many rows to look up. So let me drop that index and create one that has the include columns necessary, ClientID, Amount, TransactionDate, and TransactionID, and run the query again, and this time we get our Index Seek operation that we want. And have a look at the Output List of that Index Seek. We've got all the columns in the select clause listed there.

Filtered Indexes

There's one more thing I'd like to talk about in this module and that's filtered indexes. What a filtered index allows us to do is put a WHERE clause into the index definition to define what rows from the table are in the index. This is the only time where an index might contain fewer rows than the table has. Now this has to be a simple predicate, no ORs, no complicated expressions. This can be very useful on tables where the majority of the data is not interesting to some or most queries. An example would be a table that's using soft deletes. It'll usually have a column called is deleted that's a bit, or perhaps a date/time column called date deleted or something like that. The point is that some of the rows in the table are considered to be deleted, and hence, have no interest to the majority of queries. What we can do in that case is filter the indexes so that the rows which are considered deleted don't appear in them. Another use for filtered indexes, which I don't want to go into detail on is in creating somewhat complicated constraints. A requirement that I've seen a few times is that a column should be optional, but if a value is specified, it has to be unique. This can't be done with a normal constraint. A normal constraint considers nulls to be equal to one another, and hence, such a column can have one and only one element, which is not usually what people want. With a unique filtered index, however, the index can filter for non-null values and any number of nulls are allowed. It's not all good, of course. It seldom is. Filtered indexes have a limitation in that the predicate and the query must match or be an easily- identifiable subset of the index predicate for the query to be able to use it. If the query is parameterized, which is the norm and which is a really good thing to do in general, then the filtered index can't be determined to match the query predicate, and hence, gets ignored. Filtered indexes definitely have their uses. Unfortunately, they're fairly limited uses though.

Demo: Filtered Indexes

So in this last demo of the module, I want to show you when filtered indexes are useful and when they just don't work. And again, we're filtering on ClientID and TransactionType. Let's say that the filter for TransactionType = D is a very common one. Lots of the queries in our system filter for TransactionType = D and not other transaction types like W, S, B or whatever else we have in that column. I might be tempted to create a filtered index for that, an index which only includes the transaction types of D. If we look at our query at the moment though, it's doing a table scan. Again, we've got no indexes at this point. So let's create an index on transactions on ClientID, and this time we're going to add a WHERE clause to the index, filtering it so that it only includes the rows that have a TransactionType of D. And if I run my query now, I've got an Index Seek on that newly created index. Have a look at the properties though. The Seek predicate only shows that I'm seeking on ClientID. It's got no mention of TransactionType, and it won't because TransactionType isn't actually in this index, it's used to exclude rows from this index, so there's no way we can seek on TransactionType, it's not a key. We don't need to though. We're filtering for TransactionType of D and the index only contains rows with TransactionType of D. And so the filter is implicit. It's not, however, shown anywhere in the properties. That can get confusing, especially if you're working with an unfamiliar system. The problem here, though, is that most of the time we don't write queries with hard coded embedded literals in them. Most of the time, we parameterize our queries. So we have our query filtering for @ClientID and @TransactionType and we pass them values at runtime. And if we run this query, we're doing a table scan again. The filtered index can't be used here because at optimization time the value of D is not known. And a bigger problem, this plan is cached. It's cached and it will be reused, so even if D is the TransactionType passed on the first execution, it might not be the TransactionType passed on the second execution or the 25th or the 307th, and we can't possibly allow a situation where the query returns incorrect results. And so if the TransactionType cannot be determined to be hard coded embedded in the query and unable to change, the filtered index simply can't be used, which is quite a major limitation and does dramatically restrict their usage, unfortunately

Nonclustered Index Summary

One final thing that I'd like to touch on before the end of this module, and that's regarding the number of indexes. It's a question I get from time to time. How many indexes should I create on a table? Honestly, it's a question that can't be answered. In short, the correct number of indexes on a table is

the number that is needed to allow the workload against that table to run optimally and not one index more. There is no reason to create indexes just in case they might be needed at some point in the future. There's definitely no reason to create an index per column. I've seen that so many times, and it's a really bad idea. It's a bad idea because in general, wider nonclustered indexes are more useful as you would have seen earlier where we looked at how to index from multiple queries. The other reason that an index per column is not usually useful is because they're usually created with no analysis and no considerations to whether they're useful to the workload or not. We need indexes, so somebody goes and creates an index per column and says, indexes are created. Good job. This concludes the basics of nonclustered indexes. I could talk for a couple more hours on this, but most of that's out of scope for this particular course. So in this module, we've seen what nonclustered indexes are and why they're important. We've seen how to design indexes to support common queries, queries with equality predicates, queries with inequality predicates. We've seen the differences between indexes which support predicates combined with ANDs and ORs. We've seen which join types benefit from indexes and how. And we've seen how include columns make indexes more useful by removing the need to do Key Lookups. Finally, we looked at filtered indexes. In the next module, I'm going to take a look at indexed views for the times when you need indexes on more complicated transformations of the data that our filtered indexes allow. See you there.

Designing Indexed Views

Introducing Indexed Views

Hi, and welcome to this module on Designing Index Views. This is part of the Designing and Building Indexes in SQL Server course. I'll start this module by discussing what index views, or materialized views as they're also called, are. We'll look at the limitations and restrictions, and unfortunately, there are a lot of both. And finally, we'll look at where they're useful because, to be honest, they're very much a niche feature. I've used indexed views perhaps three times in the last eight years. What is it that makes index views interesting? Normally, a view is just a saved select statement, and when a query that uses the view is run, during the pausing phase, the view's name is replaced by the view's

definition and the resulting query is simplified. By the time the query reaches the query optimizer, there's no trace left of the view. It is, however, possible to create a clustered index on a view, not on every view, but with a properly-crafted view that meets the requirements, it is possible. This materializes the view. Its results are now stored in the database as if it was a table. It's possible at this point to create some nonclustered indexes on the view as well if necessary. The saved results aren't static as the data in the underlying tables change, so the index view is kept up to date. This does mean that the index view adds overhead to data modifications, and as such, it needs to be monitored to ensure that it's not doing more harm than good.

Indexed View Limitations and Uses

The main problem with index views is getting them to work at all. There are a huge list of things that can't be in the view in order for it to be indexable. The reasoning behind the limitations is to make it possible to determine when to change, add, or remove a row from the view's results when the data in the underlying tables change. That doesn't make it any easier to use them though. Last time I created an index view, even after having read over the full list of limitations earlier, I still had to modify the view four times before I could create the clustered index successfully. I'm not going to go through all the limitations. They're well documented, and I'll put the link into the last module, but I'll go through some of the more common ones that you're likely to run across. I've got here what looks like a reasonably simple view that you might want to index. This, however, is not indexable for a number of reasons. The first thing I need to mention, and this is a requirement not a restriction, is that the view must be created with SCHEMABINDING. This means that the underlying tables cannot have their schemas modified while the view exists. This sounds quite harmless, but it could actually end up complicating deployments. Instead of being able to change the data type of a column, you'd first have to drop the index on the view, drop the view, make the change, and then recreate the view and its indexes. And recreating those indexes can take a lot of time if there's a lot of data in that view. The first thing in this view that makes it unindexable is the c. *. For a view to be indexable, the columns must be listed explicitly. The use of * is not permitted. The second thing is that subquery. To be indexable, a view may not contain any subqueries. The third one is the outer join. This is commonly the most problematic of the limitations from what I've seen. For a view to be indexable, all of the joins in it must be inner joins. This can be quite a difficult problem to get around, especially since another restriction is

that index views can't reference other views, only tables directly. Then we have the derived table. As I'm sure you've guessed by now, that's not allowed either, nor is the MAX function inside it. MIN wouldn't be allowed either. Technically, ROW_NUMBER is permitted, but the OVER clause isn't, which makes the fact that the function itself is permitted a little meaningless, ROW_NUMBER must have an OVER clause. Soft joins are not permitted, neither is apply, table-valued functions, union, distinct, or grouping sets. I sometimes joke that the list of things that are allowed is shorter than the list of things forbidden. Unfortunately, it's actually not a joke. There's another requirement. This one's not too bad most of the time. There's a number of session settings that have to be set in a specific way for the index to be created on the view. This is the list, and this is how they have to be set. A problem sometimes arises in the fact that the specific settings are needed not just when the view is being created or when the index on the view is being created, but they needed to be set this way when the tables that the view references were first created. And one of those settings is not the default that Management Studio sets the session options for. This can be very frustrating when trying to create an index view. You go through all the limitations on the view and then find out that one of the larger tables of the view queries was created with ARITHABORT off. Recreating large tables in a production system is not usually an option. So where are index views even useful? Over the years, I've found two places where they can be beneficial. The first is for making simple aggregates fast, not, for the most part, for an analytic system. For that, we've got columnstore indexes, which I'll be talking about in the next module. But consider an OLTP system where a query that runs quite often has to aggregate a large table and it has to return that result very fast. That is an excellent use for an index view. Essentially, it's preaggregating the data. The downside is that the view has to be kept up to data, and that adds overhead, so it's not a free lunch. Another use I've seen is when a common, frequently-run query has to fetch data from one table filtered by a column or set of columns in another table. For example, we might only want to see shipments from the clients in a particular location. The materialized view removes the need to do the joins every time at the cost of adding overhead to the data modifications. I should emphasize these are not the only possible uses for index views. Just ones that I've found particularly suited for index views and accommodating for most or all of their limitations.

Demo: Creating and Using Indexed Views

In this demo, we're going to have a look at the two uses that I discussed for index views, and I'll also show you a rather nifty feature that SQL Server has, admittedly only in Enterprise edition. What I've got here is a commonly-ran query in my little test application. I've got a Shipments table and a ShipmentDetails table, and the details contains things like the amount of the volume in the number of containers per item in the shipment, but I often need to know the total mass, the total volume, or the total number of containers at the shipment level. And for this little OLTP application, this query's a little bit too slow. It's not horrible, but 400. milliseconds to run the whole thing is a little much, and there's only 26, 000 rows in the Shipments table. As this data grows, this is going to get horrible. This is a perfect use for an index view. So what I've got here is the same query and a view WITH SCHEMABINDING and the UNIQUE CLUSTERED INDEX for it. The first index that you create on a view has to be unique clustered, after that, you can create nonclustered indexes that don't have to be unique, but this one has to be a unique clustered. And since I'm grouping by ShipmentID, I know that my ShipmentID is going to be unique and so I can put my unique clustered index on the ShipmentID column. One thing to note here is that I've got a count big column in this view. I don't actually need that, not for my query, but that's another requirement of index views. If you have any aggregation, any sums, any counts, then you must have a count big * in that view as well. And if I run a simple select * from my index view, I can see that my CPU time is 0. This is massively faster than my original query was because we're not doing the aggregation. We're simply reading the data out of the existing table. It's not actually a table, it's a materialized view, but the optimizer treats it as a table. Now there's one really cool thing I'd like to show you. If I have a query which has the same form as the index view, but doesn't reference the index view like this one, which has a similar mass and a similar volume, but doesn't have some of the containers, that groups by ShipmentID, but doesn't group by ClientID. And I run this query. Something very interesting happens. This ran in 30. milliseconds, but to aggregate that entire table takes 300. We saw that earlier. We look at the execution plan, it's not going to the tables. The optimizers figured out that the index view can be used to execute that query even though that query doesn't explicitly reference the index view. And so the optimizer goes and reads the data from the index view instead of going to the base tables. Unfortunately, this is an Enterprise-only feature. In Standard edition, this will go straight to the tables, but in Enterprise, it's awesome because you can create an index view to support a query that you can't change, and the query will automatically start using the index view if it's appropriate. Technically, index views are Enterprise features only. You can

create them on Standard Edition, you can use them on Standard Edition, but you'll probably find that you need to use the WITH (NO EXPAND) hint on the query so that the optimizer doesn't go back to the base tables and does actually use the index view. The second thing that I said index views could be useful for is this kind of query. I need to filter my transactions, but I need to filter them on columns that are not in the transaction table. They are multiple joins away in the StarSystems table. Now this query isn't particularly slow. It runs in 16. milliseconds, but these are quite small tables. I can create an index view on this, and I can create my UNIQUE CLUSTERED INDEX on the InvoiceNumber column, and then when I run my query, it's now running in 0 milliseconds instead of 16. This isn't a massive saving, but as I said, these are fairly small tables.

Downsides of Indexed Views

I've mentioned at a few points that there are disadvantages to index views above and beyond their limitations. The first is that index views add additional overhead to data modifications. This probably isn't too bad unless you have several index views built on top of busy OLTP tables, but it is something that you need to keep in mind and it is something that you should monitor after creating an index view. The second is space. The views are materialized. They take up space in the data file and they take up space in memory. If you materialize views that return many gigabytes of data, the database is going to grow. The backups are going to grow and get slower, the restores are going to get slower, the consistency checks are going to get slower, and your index maintenance is going to get slower. And your DBA is probably not going to be overly happy. Speaking of index maintenance, the third major disadvantage is that the SET options that I mentioned are needed to create the index on the view are also needed when we're building the indexes. I've seen more than one case at clients where an index view was created, and suddenly, the index maintenance job starts throwing errors, and SQL Agent's error handling is not necessarily the cleanest. It's often very hard to tell why your index maintenance is suddenly failing out of the blue. In this module, I've looked at what indexed, or materialized views, are. We've discussed the long, long list of limitations and restrictions, and we've seen a couple of places where they're useful. In the next module, I'll be talking about columnstore indexes, a different type of index entirely. There is a massive benefit to analytics- type systems. See you there.

Designing Columnstore Indexes for Analytic Queries

Introducing Columnstore Indexes

Hi, and welcome to this module on columnstore indexes, part of the Designing and Building Indexes course. We're going to start this module looking at what columnstore indexes are because, to be honest, they're quite different to the indexes we've seen so far in this course. We're going to look at their architecture, including the difference between clustered and nonclustered columnstore indexes, and then we'll spend some time looking at when and how columnstore indexes should be used. Columnstore indexes were added in SQL Server 2012, although, to be honest, they weren't all that useful in that version. They've been improved in every version since then, and by SQL Server 2016, they're very functional and exceedingly useful. Unlike in a traditional rowstore index where all the values which make up a row are stored together, in a columnstore index, the values that make up the column are stored together. This makes them very easy to compress since it's quite likely that a column will have repeated values in it. They're very good for analytics-type queries, queries where you might want to sum or average or otherwise aggregate large ranges of a small number of columns. So for a quick refresher, this is at a high level what a rowstore index architecture looks like. Rows stored together on _____ AK pages built into a B-tree with multiple levels. The index key defines the logical order of this index, and we can navigate down the tree searching for values based upon the index key. Columnstore indexes are quite different. The Create Index syntax looks similar, but if go and create a columnstore index on Amount, TransactionType, IsPriority, and ShipmentID, I don't get a bunch of pages with all four stored on it. What I get instead are individual columns made up of those pieces of data, and the columns are stored together. Now, you might be asking, how do we reconstruct a row when all the data in the columns is stored together? The data in these columns is not sorted in any way. It's as they're read out of the table, so if I want to reconstruct a row that has an amount of 80.23, that is the seventh row in this table in the order that we read the data. So I get the 7th row from the Amount column, I get the 7th row from the TransactionType column, which is C, I get the 7th row from the IsPrority column, that's 1, and I get the 7th row down of the ShipmentID column, that's 8. And

that's how you'd reconstruct a row with a columnstore index. These groupings of columns are then further divided into rowgroups, of it mostly in rows. Each segment, which is one column from a rowgroup is then compressed and the resulting compressed data is divided over the AK pages that we're familiar with for storage purposes. There's a lot more to the compression than that, but I'm not going to go into detail here. It's quite out of scope for this course. And in the last module, I'll include some links on further reading on columnstore indexes. Now this isn't the entire story because in this form, a columnstore index is read-only. Rows can't be easily deleted from a columnstore because of the compression, and single-row inserts would be really inefficient if they required a recompression after every insert. What's done then is to essentially make the compressed portions of the columnstore index read-only. And to add to that index a deleted list, which is a tree much like we've seen before and what's called a deltastore, which is itself a B-tree. When a row is deleted from a columnstore index, or updated, the position for that row is flagged in the deleted list as deleted, meaning that when the index is read by the query processor, those rows are ignored. The deltastore then contains newly-inserted rows and the new values for updated rows. And this is a very traditional B-tree. These two trees automatically manage behind the scenes for us. I'm not going to go into a huge amount of detail. I'll show you some of it in the demo, and I'll include some links to further information in the last module. Lastly, there's two types of columnstore indexes much like there are two types of rowstore indexes. We've got the clustered columnstore index, which essentially is the table. It does the same job of organizing the table as the clustered rowstore index does, and like the clustered rowstore, it consists of all the columns in the table. Then we've got the nonclustered columnstore indexes, which are secondary structures. These contain just the rows that are specified much like nonclustered rowstore indexes. And from SQL 2016 onwards, you can have clustered columnstore indexes with nonclustered rowstore indexes. You can have a clustered rowstore index with a nonclustered columnstore. You can have a heap with nonclustered indexes of both kinds.

Demo: Creating Columnstore Indexes

In this demo, I'm going to create a couple of columnstore indexes on a large table that I have generated, and I'm going to show you some of the metadata behind it and how the index is affected by rebuilds and reorganizers. I've got a copy of my ShipmentDetails table without any indexes on it, and I'm going to use that to create a clustered columnstore. But before, I'd like to show you the size of the

table without a columnstore. If we run this query against these indexes, you can see that the table at the moment is a heap. It's a table with no clustered indexes on it at all, and it's got just over 10, 000, 000 rows in it. This query against sys.dm_db_partition_stats shows the row_count of the table among other things. The interesting things in this case are the page counts. They show us how big the table is. A page, if you remember, is 8 KB. So let me create a clustered columnstore index on this table. Note that I'm not specifying columns here. Columnstore indexes don't have keys, and since this is a clustered columnstore, it implicitly includes all columns in the table, and so I don't need a column list. Just create CLUSTERED COLUMNSTORE INDEX, index name, and the table. The only time you need column lists for columnstore indexes is when you're creating nonclustered ones. But if we look at sys.dm_db_partition_stats again, you'll see the table is now dramatically smaller, 14, 000 pages. We're roughly a third of the size that we were as a heap. If I go back and run my query against sys.Indexes, you can see that the table now has a single index of type CLUSTERED COLUMNSTORE on it. I'm going to drop that clustered index, turn the table back into a heap, and then create a nonclustered columnstore in the same table. I'm not creating this on all of the columns this time, just four of them, Mass, Volume, Containers, and ShipmentID. And if I go back and look at sys.Indexes, the table itself is still a heap, but now I've got a nonclustered columnstore index on that table. Just as with nonclustered rowstores, this is a separate structure from the table. And if we go and look at sys.dm_db_partition_stats, the nonclustered columnstore is 11, 000 rows. It's smaller than the clustered columnstore because it's missing some of the columns that the clustered index has. I'm going to drop the non-clustered index and create my clustered columnstore index again. And now I'd like to show you another system table around columnstore indexes, specifically the one that shows the rowgroups. And if you look at the results from sys.column_store_row_groups, I've got 17 rowgroups here. This table's got 10, 000, 000 rows in it, and hence, with rowgroups that should be a million rows each, ideally I'd only have 10 rowgroups, but I've got 17. Some of them have a million rows in them, but some of them are as small as 45, 000. There's a number of reasons why this could happen, and I'll include some links in the final module with more information on this subject. What I want to do now is delete a bunch of rows from this table, insert a bunch of rows into this table, and then update some rows in this table so we can see how it affects the columnstore internal stats. If I look at my rowgroup system table again, I've got another rowgroup. This one, however, has got a state of open. This is my deltastore. This is the B-tree that contains the newly- inserted and new values for the updated rows for

this table. And if I carried on inserting and updating, this rowgroup would grow in size until it reached a million rows, at which point something called the tuple mover would come, pick up that rowgroup, compress it and turn it into a normal rowgroup in the columnstore index. I'm not going to do that in this case. What I'm going to do is rebuild my index because that forces the compression of all open rowgroups. If we go and look now, I'm down to 15 rowgroups. I've still got some very small ones though, 8, 306 rows is a little on the small side for a rowgroup for a columnstore index. Rebuild doesn't do a lot of merging of rowgroups. What does the merging of the rowgroups is ALTER_INDEX REORGANIZE, so we run that now, and go back and look, and now I've got the 10 rowgroups that I was expecting to have upfront, most of them with a million rows in and then the smallest one having 300, 000. So that's a look through the system tables and some of the internal details of the columnstores. Now we'll get on to actually using them for our queries.

Where to Use Columnstore Indexes

So that was an interesting look at the architecture of a columnstore index. Where architecture is only part of the question, we want to know where these things are useful. They're exceedingly useful in analytics systems, in data warehouse-type systems. As clustered indexes are the large fact tables. This means that your fact tables are heavily compressed, which gives you space benefits, data warehouses typically being really large, and it means that your aggregations or your fact data is exceptionally fast. If the data warehouse is also doing a lot of searches, searches for individual rows in the fact tables or small sets of rows in the fact tables, then you can add nonclustered rowstore indexes to the clustered columnstore to help with those searches. The nonclustered columnstore index is quite useful when you're doing analytics in what is otherwise an OLTP system. What we'll do in this case is optimize the table for our OLTP-type queries with a clustered index that either organizes a table or supports a common access path as was discussed in a previous module, and then we put a nonclustered columnstore onto columns that are frequently aggregated so that we can get very fast analytics without seriously impacting our OLTP system performance. Now as with any index, there are some considerations for when you want to use columnstores. The first thing to consider is the columnstore indexes are not ideal for systems that have lots of single-row modifications, that's updates or inserts, especially updates. What I said on a previous slide, you can use nonclustered columnstores to support analytics and OLTP systems. If the OLTP system has a lot of high- frequency

inserts and updates, your columnstore might not work very well. And this comes down to the deltastore. A lot of the benefits for columnstore indexes is in their compression, the fact that they are massively compressed and the fact that you read entire columns at a time. But the deltastore, which is where our newly-inserted rows and new values for updated rows go, is not actually a columnstore. It's a stock standard traditional rowstore B-tree. And if you've got a system where there's lots of frequent inserts and your queries are looking for the recently inserted rows, the columnstore might not help you because a lot of the rows you're filtering for aren't in the columnstore itself, they're in the deltastore. The deltastore's not compressed, the B-tree is not going to be built on an ideal set of columns for seeks, and so you're getting the worst of both worlds here. You can't seek for your data and you don't have the compression for the columnstore. If the inserts are faster than the background tuple mover can handle, you could actually see the deltastore constantly growing and not getting turned into actual columnstore rowgroups. The second consideration is that columnstores are not efficient on small tables. This again is coming down to compression, the compression benefits from having lots of rows in a single rowgroup, ideally a million. This isn't just something that catches people out on small tables though. If you've got a large table that's partitioned and those partitions are very small, that's also not beneficial to a columnstore because those partitions are essentially separate tables and so your columnstore's rowgroups don't have as many rows as they should have. You probably don't want to put a columnstore index on anything that has less than a million rows, and, to be honest, I'd probably start looking at columnstores once I've got 10, 15, 20 million rows in a table. The compression, which is one of the key features of columnstores works best on columns with repeated patterns in the data. If you're encrypting data, don't put a columnstore index on that please because it can't compress. The encrypted data is random, and so the columnstore's got virtually no compression, which means it's throwing a lot of its benefits out. This also goes to things like discussions, large string columns, columnstore indexes are ideal for numbers, for columns with repeating patterns. They are not particularly good for columns that have got a lot of very distinct data, very random data, or encrypted data in them. Because they don't have keys and they don't have B-tress, a columnstore index cannot be used to seek. So if you've got a lot of singleton seeks or small-range seeks on a table, columnstore index is not going to help you here. You may want a combination of columnstore and rowstore indexes for that pattern, but again, it's going to depend on what you're doing. What I'd like to emphasize here is that columnstore indexes, while they are awesome, are not a replacement for our traditional

rowstore indexes. There's a place for both of them. And if I'm going to be doing a lot of single-row seeks against a table and very few large-range scans with aggregation, I probably don't want a columnstore index on that table. And the last one is that you can't use a columnstore index to enforce uniqueness, so you can't create your primary keys or columnstore. You can't create your unique constraint to the columnstore. If you need to enforce uniqueness on a table, and in many cases you should be enforcing uniqueness on a table, you're going to have to use a rowstore index for that. Data warehouses can sometimes get away without enforcing uniqueness because the assumption there is that the uniqueness was enforced in the source system, and the data warehouse is a copy of the truth. But your OLTP systems, your front-end systems, your actual line of business systems, you probably want to enforce uniqueness on those tables, which means you're going to need a rowstore index for your primary key, your unique constraints, and your unique indexes. And it's worth noting at this point that if you'd like to use foreign key constraints, a foreign key constraint requires that the parent table has a unique constraint, unique index, or primary key on the column being used. So if you're going to go for columnstore indexes without unique constraints, you're also not going to be able to use foreign key constraints with all the implications thereof. Now you might be asking yourself at this point, but what about the order of the columns? Because we spent a lot of time looking at the order that the columns should be in with nonclustered rowstore indexes, but I haven't even mentioned that for columnstores. And that's because, to be honest, it's not a matter of major importance. A columnstore index is not something we seek on. It's an index designed for scans. And so the important thing with a columnstore is is the column in the index, not where is it in the index? There's no concept of left-based subset of the key because columnstore indexes don't even have a key. It's not a tree. We're not seeking. We're not navigating. So all of that left-based subset and equality before inequality that is so important for rowstore indexes is meaningless for columnstores. What is important for the columnstore indexes, for the nonclustered columnstore indexes, is to ensure that all of the columns that the queries need are in the index. So if you've got a query that groups by three columns and aggregates two more, all five of those columns need to be in the nonclustered columnstore index. In which order? It really doesn't matter. They just need to be in the index. For clustered columnstores, you don't even specify a column list at all. It's a clustered index, meaning it contains every column in the table, and since columnstore indexes don't have a key, there's no column list defined for a clustered columnstore index as you'll see in the upcoming demo.

Demo: How SQL Server Uses Columnstore Indexes

So in this demo, I'm going to create a columnstore index on my table and show how it benefits query performance for aggregate queries and how it doesn't benefit query performance for queries which are seeking for a small number of rows. I've got here a query that probably looks quite familiar. It's similar to the one that I used in the previous module to show the power of index views. This time I'm going to use a columnstore index to make it faster rather than an index view. And if I run this query as it is at the moment, on a heap, no indexes at all on this table, it's taking about 12 seconds of CPU time and just over 2 seconds of a lapse time to execute. I still have 26, 000 rows been returned, but my details table has a lot more data in it. It's got 10, 000, 000 rows in it. So first thing I want to try here is a clustered columnstore index on that table. That's all of the columns in the table going into the columnstore. If I run this again, you can see the massive reduction in execution time. My CPU usage is down under a second, and my lapse time is under half a second. That's just from adding a cluster columnstore index on that. I've done nothing else to this. If we look at the execution plan, you can see a slightly different operator this time, Columnstore Index Scan. And if we look at the tooltip, most of the things here are familiar, and we can see the output list returning the columns that the query was interested in. The Hash Match aggregate is doing the group by and then the Parallelism (Gather Streams) is merging the different parallel streams so the data can be returned. There's one other thing that's important here on the tooltips though. The columnstore index is exceptionally good at making aggregate queries fast, but it's not the entire story. The Execution Mode here is also really important. If you'd noticed the Execution Mode in any of the previous modules, it would have been Row. Here we've got Batch. Batch is new mode of execution for SQL 2017, limited to columnstore indexes. And we aren't going into a whole lot of detail. Batch mode allows for more than one row to be processed at a time by the query processor. The Batch mode is actually the key to making queries using columnstore indexes really fast. And I should mention that in SQL 2019, which at time of recording is still in CDP, Batch mode is no longer limited to columnstore indexes. You can get Batch mode on rowstore indexes as well, which is an awesome improvement. But back to SQL 2017, my clustered Columnstore Index Scan is in Batch mode, my Hash Match aggregate is in Batch mode, and then my Parallelism (Gather Streams) is in row mode, and that's fine because by this point the aggregation's done and we're down from 10, 000, 000 rows to 26, 000 that have to get put together to be returned to the user. So let me quickly drop my clustered columnstore and create a nonclustered columnstore.

And to start with, I'm going to put that on Mass, Volume, Containers, and ShipmentID. If this was a rowstore index, this would probably not be the most optimal column order because I'd want the ShipmentID first because that's what I'm grouping by, but it doesn't matter for a columnstore index. This is in the key. Go back under my query. It's improved again, slightly. You shouldn't expect huge gains going from a clustered columnstore to a nonclustered columnstore, and indeed, we're not seeing huge gains. The CPU time's, in fact, very slightly higher. The lapse time is very slightly lower. I'd have to do a lot more tests before I would say that was something to expect or just an artifact of other things running on this machine. If we look at the execution plan for the query using the nonclustered columnstore, it's just about identical. The only difference here is that our Columnstore Index Scan is a nonclustered columnstore instead of a clustered, but that is about the only difference we can see here. Let me drop that index and recreate it with a different column order, ShipmentID first, which for a rowstore index is what I'd expect. If I run my query again, the execution plan is identical. We still have a columnstore index scan, we've still got the same outputs, we've actually changed nothing, and that's because the column order for a columnstore index doesn't matter very much. This isn't an index tree. We're not navigating down the tree, and so the order of the columns in the index doesn't have that much meaning. One more thing I'd like to show quickly, and that's a query that's filtering for a single ShipmentID. We'll run that and look at the execution plan. I've still got a columnstore index scan. ShipmentID is first in that index, but again, this isn't a B-tree, this isn't a seekable index, and so we're still doing an Index Scan to fetch this data even though we're fetching 1, 298 rows from 10, 000, 000 because this is a columnstore we can't seek. If you've got a lot of small or singleton seeks, you'd probably want to create a nonclustered and rowstore index on the same table, whether it be with a clustered columnstore or with a normal clustered rowstore index on the table itself. In this module, we had a look at columnstore index architecture, seeing how it differs from traditional rowstore index architecture. And we had a look at the guidelines for creating columnstore indexes, where they'll be useful to your systems, and where you should probably think twice about creating them. In the next module, the last module for this course, I'm going to wrap up everything I've covered in this course and then provide some suggested courses for further study and some links to further information on things I've discussed in this course. See you there.

Summary and Further Reading

Course Summary and Further Reading

Hi, and welcome to this last module in the Designing and Implementing SQL Server Indexes course. In this module, I'm going to summarize everything that we've covered so far and then provide some suggestions on further courses and additional reading. In this course, we've covered clustered indexes, we've covered nonclustered indexes in a lot of detail, we looked at index views and when they're useful, and finally, we looked at columnstore indexes. Clustered indexes are the indexes that define the table storage. They are both rowstore and the columnstore clustered indexes, and what you choose will depend upon how your table is used. Your rowstore indexes are excellent for OLTP systems, for systems with lots of small, fast queries. Columnstore indexes are excellent for analytics systems where you're frequently reading large portions of the tables to aggregate the data in some way. When using rowstore clustered indexes, the two options that you've got are to choose the key to organize the table or to choose it for the most frequent access path. If you choose a key to organize a table, then you're minimizing page splits, making sure that your inserts and updates are fast and ensuring that your table isn't larger than it needs to be. If you're indexing for the most frequent access path, you might be making tradeoffs in terms of page splits and certain update speeds or table size in order to gain read speed for a query. Our nonclustered indexes are separate structures from the table and hence, we can have multiple of them, the current limit being 999. Please don't try that. I really don't want to see a table with 999 nonclustered indexes on it. Like with a clustered index, you can choose rowstore or columnstore nonclustered indexes and in the latest versions of SQL Server, you can create both on the same table. You can create rowstore nonclustered indexes on a table that has a clustered columnstore, you can create nonclustered columnstore indexes on a table there's a clustered rowstore index, you can create both rowstore and columnstore indexes on tables. There isn't that much use in creating a nonclustered columnstore on a table that has a clustered columnstore. There might be an edge case that you find that is useful for that, but generally, your nonclustered columnstore indexes are good for tables which have clustered rowstores. For your rowstore indexes,

the order of the columns in the index key should be determined by the queries that run against your system. The key things that you need to remember here are left-based subset of the index key and equality predicates before inequality predicates. Keep those two principles in mind and you'll find that your indexes are useful for the queries that execute on your systems. Your nonclustered columnstore indexes are useful for analytics queries running against operational tables. That would be tables in OLTP systems. The column order doesn't matter here. All that really matters is that the columnstore index contains the columns that will be used for these analytics queries. An index view forces a view to be materialized and stored as if it was a table. That means that the query forming the view doesn't have to be executed every time the view is queried. There are a lot of restrictions on what's allowed in them, which does tend to make creating index views somewhat of a hit and miss affair. You'll typically run into one or more of the restrictions most of the time. They are quite useful for queries that often aggregate data, especially if it's aggregations on a single table. That way you're not running into the restrictions on outer joins. They also can be useful for queries that need to filter one table based on data in another table several joins away. So that's a summary of everything we've covered in this module. If you're interested in studying further, I highly recommend the Indexing for Performance course by Kimberly Tripp. That is 8 hours on indexing. Kimberly is absolutely phenomenal as a teacher. She taught me a lot about indexes, and this course goes into way more detail than I could. It's a deep dive on SQL Server indexing. And then if you're looking for more information on query performance, execution plans, or anything in that area, Erin Stellato's course on Analyzing Query Performance for Developers is also excellent and she goes into a huge amount of detail and execution plans on finding slow queries and on analyzing queries. The SQL Server Documentation on Indexes is comprehensive and detailed, and it contains all the information on the various keywords and options that you have for creating indexes. I highly recommend giving that a read before you dive into indexing your systems. There's a lot nuance that I haven't been able to cover in this course, which the documentation will touch on. There's the Index Architecture and Design Guide, also in the Microsoft Documentation. This is also exceptionally good on when to create indexes and how. There are two specialized types of indexes that I haven't even mentioned in this course. There's your XML indexes, which are created on XML data types to allow the XPath queries to execute efficiently. And if you're interested in those, again, the Microsoft Documentation, XML Indexes in SQL Server, is quite good. The second type of special index that I haven't mentioned is the spatial indexes. These are used when

you've got geometry or geography data types and your query needs to determine whether one piece of geometry intersects another one, is near another one, or any of those kind of operations. Again, the Microsoft Documentation is quite comprehensive on this subject, so give that a read if you're playing with spatial indexes. For a full list of the index view limitations and requirements, you can consult the Microsoft Documentation page. They're listed very clearly there. Onto columnstore indexes. And it comes with a whole pile of articles on columnstore indexes. At the time of this recording, he was up to 127 parts of his series on columnstore indexes. Columnstore index dictionaries, how they're created and how they're maintained, is in part 13 of his series. And for the Causes of Small Rowgroups, that is rowgroups under the million rows, you'll find a lot of information in part 62 of his series. I do recommend reading the other 125 entries when there's time. They are comprehensive, detailed, and very interesting. And finally, on Columnstore index maintenance, there's the Microsoft Documentation on when to reorganize, when to rebuild, and what both of those do to columnstore indexes. Finally, there are two blogs that I'd like to call out. Firstly, my blog. I've got a whole pile of articles on indexes. I've got a few more that I'm planning to write, but writing time's been a little limited recently. I do promise to write some more on indexes soon, whenever soon is. The other blog post, which is of massive interest is Kimberly Tripp's. She doesn't write much lately, but what is there is excellent, detailed, and very comprehensive. Thank you for watching this course on I do hope that you've learned a few things, and I hope that you have a lot of fun and a lot of success in indexing your systems and gaining good performance improvements out of your systems. I hope to see you in a future course. Goodbye.