Course Overview

Course Overview

(Music playing) Hi everyone! My name is Pinal Dave, welcome to my new course, Analyzing Query Plans. I am a SQL Server Performance Tuning Expert at sqlauthority.com. To solve any problem, it is critical that we have the correct root cause of it. Similarly, if you have slow performing SQL Server queries, analyzing query plans is like finding root cause of your poorly performing queries. This course gives you a comprehensive view of major components of analyzing query plans. You just need a basic understanding of SQL Server for this course. Some of the major topics that we will cover are how to capture query plans using extended events and traces, identifying poorly performing query plan operators, how to create efficient query plans using query store, how to compare estimated and actual query plans, and configuring Azure SQL database performance insight. By the end of this course, you will have a solid foundation to get started with analyzing query plans. During the course, along with finding the root cause of slow performing queries, we will also learn how to resolve major performance bottleneck effects as well. I hope you will join me in this journey to learn Analyzing Query Plans at Pluralsight.

Capture Query Plans Using Extended Events and Traces

Version Check

Introduction

Hi, this is Pinal Dave, and I welcome all of you to this course, Analyzing SQL Server Query Plans. In this module, we will discuss about how to capture query plans using extended events and traces. Let's see an agenda of this module. First, we will discuss about query plans. Right after that, we'll have a brief introduction about SQL Profiler. Next, we will discuss about SQL traces, and right after that we will explore SQL extended events. We will see a demonstration of each of these concepts. Now, let's start learning about query plans in the next clip.

What Is a Query Plan?

Russell Ackoff said, "We fail more often because we solve the wrong problem than because we get the wrong solution to the right problem. " It is extremely critical that we find the root cause of our problem. Sometimes we find a solution to the problem which does not exist and we end up in a very awkward situation where we still have the original problem and help solve the problem which does not exist. This is extremely relevant to query plans. Query plans help us to understand what is the root cause of slow performing query. Let's understand what is a query plan? As per Wikipedia, a query plan or query execution plan is an ordered set of steps used to access data in a SQL relational database management system. Essentially query plan is the set of instructions to reach a correct solution. Since SQL is a declarative language, there is typically a large number of alternative ways to execute any query. There can be many different plans and each plan can have a different performance. When query is submitted to database engine, the query optimizer usually evolutes various different options and find the most efficient option which brings you to correct results. However, there are some moments when query optimizer just builds inefficient execution plan. At that point in time, it's the responsibility of the DBA or query developer to find slow performing operator and find an alternative to it. Professionally I am a SQL Server performance tuning expert, and when I see a slow-running query, I get very excited, as it is always challenging to find the root cause of slow performing query and find an alternative to make that query run efficiently. Well, that's a brief note about query plans. Now let's see in the next clip query execution plans' various important stages.

Query Execution Plan Stages

In this module, we are going to discuss about query execution plans' four important stages. The very first stage is query parsing. As the SQL query arrives to relational engine, it goes through a set of checks that validates if the SQL is written correctly following all the necessary syntax. This process is known as query parsing. During the parsing, the relational engine builds the logical steps to execute that query. The very next step is the query optimization. The query optimizer takes the logical steps built by the query parsing process and various relevant data and apply the optimizer model. The optimizer model has the vision to build an execution plan, which is the most optimal way to execute any query. The query-specific optimize model, which is generated by the query optimizer, is called an execution plan. In SQL Server, the query optimizer uses cost-base plan. A cost-base plan is where the optimizer decisions are based on the cost of resources and efficiency of the query execution. The next step is query execution itself. Once the execution plan is generated, the query execution phase starts where the relational engine interacts with the storage engine to fetch and update the necessary data. At the end of this phase, storage engine returns the result of query execution to the client. While this phase is executed, there are possibilities of changing the query execution plan if the statistics related to stored data are incorrect or stale. And the final phase is query plan cached. From the previous three steps, you can realize that query execution plan generation is an expensive operation. In SQL Server, query engine tries to save and reuse the same plan as much as possible. The plans are stored in the memory buffer area and is called as plan cache. When a query is submitted to relational engine the first time, it generates the estimated execution plan. At this point in time, it does not have any reference points; hence, it executes and generates the actual execution plan, which is stored in plan cache. When the query is submitted the second time or subsequence times to the server engine, once again, it generates the estimated execution plan. At this point in time, if the generated estimated execution plan is the same as stored actual execution plan, SQL Server immediately uses the execution plan which is stored in plan cache. This reusing process of execution plans saves a lot of overhead of creating new actual execution plans and helps boost your server's overall efficiency. Well, this covers four important steps of the query execution plan. In the very next clip, we will discuss why we should analyze query plans.

Why Analyze Query Plans?

Let's try to answer why analyze query plans? Well, there are five reasons why we should be analyzing query plans. The very first reason is to identify expensive operators in your query plan. When any query runs, there are multiple operators, and it's important to know what each operator does and how much resources each of them consume. If one of the operators is taking more resources than others, we must understand why it is taking and if there is any other way to optimize that. Another reason to analyze query plans is to understand data flow. There are many different operators and data flows from one operator to another operator. Sometimes there is excessive data flowing

from one operator to another operator. A good performance tuning expert would know various different tricks to control data flow and optimize it. Another reason to analyze query plans is to diagnose blocking operators. There are two different kinds of operators; blocking and unblocking. If your query is using any of the blocking operators, it's quite possible your query may run slower. This is when you need to use alternative ways to write your query so your query can avoid using those notorious blocking operators. Query plans also gives us state of statistics; older or stale statistics sometimes give us inefficient execution plans. And finally, analyzing query plans gives us classification of resource consumption. By looking at query plans, we can understand how much CPU, memory, or IO the entire query plan is consuming. If any of the resources are not available, your query plan will warn that and your query will run slower. If I had to summarize this entire slide in one word, I would say to understand the root cause of slow-running queries, you must analyze query plans. Well, so far we have talked about quite a lot of theory about query execution plans. Let's see in the next clip a demonstration where we will discuss three different types of query plans.

Demo: Execution Plans

Let's see our very first demonstration about execution plans. Over here we will see how we can enable execution plans in SQL Server Management Studio. We will see three different examples, graphical execution plans, textual execution plans, and XML execution plans. Just for simplicity, we will explore only actual execution plans in this module. Now, let's go to SQL Server Management Studio and see our very first demonstration. Here we are in SQL Server Management Studio. I will be using WideWorldImporters as a simple database. Instruction to download WideWorldImporters is already provide in demonstration folder. Now here is our very first query. SELECT * FROM WideWorldImporters.Sales .Invoices. Technically you do not need this simple database to run this query. You can run similar queries against any database and understand the concept which I'm explaining in this demonstration. First, we will see graphical execution plans. As I mentioned earlier, we'll be only exploring actual execution plans in this demonstration. To see graphical execution plans, we have to enable graphical execution plan settings in SQL Server Management Studio. We can do that via different methods. One of the methods is to type shortcut CTRL+M. I use it very frequently when I am with my customer for doing SQL Server performance tuning. But another method is to click on the icon over here in SQL Server Management Studio. You can see when I mouse over this particular icon, it shows the bubble Include Actual Execution Plan and also shows the shortcut. You can click over here and it will also enable actual execution plans for any particular batch. You can also enable graphical execution plan by going to Query and selecting this particular option. You can use any of these three options to enable graphical actual execution plan. Now I have already enabled this option, let's run this query and see the output. Select the query and click on Execute. Query is running, and after a few seconds, Execution plan tab is visible in Results set. Click over here and now we can see graphical execution plan. These are the various operators of our query execution. We will understand everything in detail in later modules in this course. For the moment, we are just learning how to enable execution plans with the help of SQL Server Management Studio for our query. If you want to see XML of this

execution plan, you can right-click anywhere in execution plan and go to Option, Show Execution Plan XML. When you click over here, it will show you various details about this execution plan in XML. You can save this query and send it to anybody via email or file sharing. When they open this file, they will see execution plan. Now the next method is to see text execution plans. Before we go there, first let's disable actual execution plan by clicking over here. This will stop display of graphical execution plan. Next, let's pay attention to the query on the screen. To see any execution plan in text format, we have to enable SET STATISTICS PROFILE ON. When you execute this statement, SQL Server will include execution plan in text format. By running this particular statement, it will turn off text execution plan. Let's run all these three statements together. Select the statement and click on Execute. At this point in time, our query is running. Once the query is executed successfully, let's go to our results set and scroll down. When we scroll down all the way to bottom, we can see various details about this query in text format. Entire execution plan is displayed over here in the results set. Let me make some space and adjust the screen in SQL Server Management Studio so we can see a little bit more detail about our execution plan. Our execution plan has four different operators, and here are details of our operators. Rows, executes, text, and various details are given right next to it. I personally use text execution plan when I want to do various comparisons and send it to my client that I worked on the execution plan and improved that performance. Details about each of the operators and workflow can be easily obtained by various columns displayed in text execution plan. The next thing is to see XML execution plan. Just so you know, we have already seen XML execution plan when we were discussion graphical execution plan. However, we can also enable XML execution plan with the help of the SQL command. If you execute SET STATISTICS XML ON, SQL Server will show entire execution plan in the XML format in the results set. Let's see how we can do that by executing these three statements. The very last statement in this query is to turn off the XML plan. Let's select them and click on Execute. Once the query is executed successfully, the very last row is our XML execution plan. When we expand this column, we can see limited information. To see them in detail, let's click on this link and it will open execution plan in graphical format in SQL Server Management Studio. Now let's go back to our XML execution plan in our results set. During a performance tuning consultation, I often ask my clients to save this particular XML and send it to me in the format .sqlplan. When they send me, I can easily open them in my SQL Server Management Studio and can understand what actually is going on in their system. This is a very important step if you want to pursue your carrier as performance tuning expert. Now we have seen three different kinds of execution plans, let's see in the next clip how to analyze query plans.

How to Analyze Query Plans?

In the previous clip, we have seen demonstration of three different kinds of execution plans. Now we will see how to analyze query plans to identify there are troublemakers operators in it who are responsible for poor query performance. Well, there are four different ways to analyze query plans. The very first method is Profiler. It's very old and famous. If you talk to anybody who is mildly familiar with SQL Server, they would be familiar with Profiler. Profiler

is one of the most favorite tools of many performance tuning experts. Personally, I'm not a big fan of Profiler and the reason for the same we will see a little bit later on. The next one is SQL traces. We will not talk too much about SQL traces in this course or this module because as per Microsoft, SQL traces are marked as deprecated feature. Microsoft may remove this feature from SQL Server anytime and they do not have to inform us. There is a better replacement of traces in Profiler and that is extended events. We will discuss how to get started with extended events a little bit later in this module. Another feature to analyze query plans is query store. We will be discussing about that in a little bit detail in module 4. Once we analyze query plans, the next step is to identify poorly performing operator and see if we can optimize that. We will also explore that concept a little bit later on in module 5. Now let's understand in the next clip a little bit about SQL Profiler and traces.

What Is SQL Profiler?

In the previous clip, we have explored execution plans. Now understand two of the important tools to analyze query plans. The very first is SQL Profiler. SQL Profiler is graphical user interface to SQL trace to monitor various events running on a SQL Server instance. This is one of the most popular tools any DBA would be familiar with. I have been using SQL Profiler for over 16 years. However, SQL Profiler is no longer my favorite tool because when my SQL Server is struggling with resources, at that particular time, SQL Profiler is not as helpful as it should be. The reason is SQL Profiler itself is an extremely resource intensive process. When you turn on SQL Profiler on the server which is struggling with performance, it further degrades the performance of that particular server. This makes it even more difficult to get actual and realistic data for analyzing query plans. I use SQL Profiler only on those servers which does not have resource issues. When your SQL Server does not have enough resources to run queries and you turn on SQL Profiler, I have seen sometime queries are taking 2x time or 3x time to run the same query which was running much faster before I turned on SQL Profiler. Instead of using SQL Profiler, nowadays I depend more and more on extended events. Now let's talk about SQL traces. SQL traces gather events at the instance level and report them efficiently to users. SQL Profiler can easily create traces and track the data. Now SQL traces was my favorite tool until Microsoft decided to deprecate them. Since Microsoft deprecated SQL traces and gives us suggestion to use extended events, I have personally stopped using SQL traces and now I focus solely on extended events. However, once in awhile I still encounter SQL traces; hence, we will see a very quick demonstration of SQL traces and SQL Profiler in the very next clip.

Demo: SQL Profiler

In the previous clip, we have discussed about SQL Profiler and now we will see a quick demonstration about how SQL Profiler works. For that, we will go to SQL Server Management Studio. Here we are in SQL Server Management Studio and now we will go to Tools. Over here, the very first option says SQL Server Profiler. Click over here and it

will bring up SQL Profiler On the very left side, click on New Trace. If you are not connected to SQL Server, it will ask you for connection. Click on Connect, and next it will bring up Trace Properties. Over here we can select out desired template or events. For example, there are a set of templates Microsoft has already built for us. We can select different templates as per our need, or we can go further into Event Selection and can select our specific event. At this point in time, I'm just going to continue with standard template. Click on Run and here we go, we have our SQL Profiler running. A quick look at various columns tells us that this SQL Profiler tracks various different information related to our T-SQL statement. Now let's go back to SQL Server Management Studio and run a workload. Here we are in SQL Server Management Studio, and I have written various select statement on different tables of a simple database WideWorldImporters. Click on Execute and queries started to run. Now go back to Profiler and over here you will notice that Profiler is tracking various queries which are executed at runtime. You can pause and stop your Profiler at any point of time. Over here you can click on any statement and can see various details. For example, we will select this BatchCompleted event and it displays that this was the query which was executed. You can further see that on the text data as well. The next column talks about application name, user name, and login which was used. Here are important columns related to resources, CPU, Reads, Writes, and Duration. For example, this particular query took around this much CPU, this many page reads, and duration was around 3 seconds. Further scrolling on the right side, we can also notice start time and end time of this query. Well, this is just a standard template and it collects various important details. This is the simplest example of Profiler, but to run Profiler, it is an expensive operation. It is always recommended that you run Profiler on remote server, even though Profiler generates a lot of network traffic and slows down your server which is struggling with performance. There is a way you can use Profiler to create SQL trace and run it on server-side. As I mentioned to you earlier that it is a deprecated method, but for the sake of completeness, we will cover that in the next clip.

Demo: SQL Trace

In the previous clip, we have seen how SQL Profiler works, and now we will see how SQL traces work and we can configure it to run automatically on server-side. Let's go to SQL Server Management Studio and open Profiler. Here we are in our SQL Profiler where we have left in the previous clip. Now let's go to File, click on Export, and go to Script Trace Definition. This will give us two options; select the very first option for SQL Server 2005 to 2019. Even though traces are deprecated, Microsoft is still supporting this particular feature, and that's why we are looking at it right now. Now, click on For SQL Server 2005-2019 and save this trace file. I will give this trace file a name, MyFirstTrace. Click on Save, and that's it, we are done. Now we have created a trace file which we have saved as SQL file. We will go to SQL Server Management Studio, and now open that particular trace file. Here is the trace file, which we have just created. When you scroll down, you can notice various different stored procedures and configurations. These are all the events which we have selected. As a matter of fact, you can take this particular file and send it via email or any other sharing mechanism to your colleague or coworker and request them to configure

the exact same trace on their machine and now they can run same trace and see the various events which you are looking at. This was one of the very convenient features when SQL traces was extremely popular. Scroll up all the way and here it is asking us a file name. Essentially, this is the file which is called as a trace file and we can now configure it to run without help of Profiler by various stored procedures, which we will see in the next few moments. First, let's give this file a destination where we will store all of our traces. I will give this as my destination location. Now let's execute this query and it should run successfully and display us trace number once it is created. Here it is. As I executed this query, it is displaying us that this trace is created on server-side and its number is 2. If you create another trace, SQL Server will keep on increasing these trace IDs. Now let's understand these three important stored procedures which are related to trace. Very first stored procedure is sp_trace_setstatus. Please note this is the ID of our trace. Our trace ID is 2; hence, I have typed here 2. The 1 stands for starting the trace and 0 stands for stopping the trace. If you want to delete this trace, you can just type in 2 at the end and it will delete this trace. Before deleting this trace, it will stop the trace automatically. Now let's execute this very first stored procedure and start our trace ID 2. This command completes successfully. That means our trace is now working behind the scenes. To validate that, let's execute our workload. Please note SQL Server Profiler is not running right now; however, every detail and everything which we are running are automatically called in our trace and stored into the file which we discussed earlier. To validate that, let's open our Profiler. Now over here we will click on Open Trace File and click on mytracedata, which I have saved a few seconds ago. Open this file and it will load the trace successfully. The trace has all the data which had run earlier. This trace is currently an active trace. To check that, let's go back to SQL Server Management Studio and click on our workload. The workload is completed, so let's run it one more time, and now open Profiler and click on Open Trace File. Here, once again select this file, and click on Open. Now you will notice it contains various data related to our recent trace. As the trace is running on server-side, SQL Profiler does not add any additional workload onto that server which is already struggling due to resources. That's why SQL traces are preferred over SQL Profiler. Let's go back to our SQL Server Management Studio and stop the trace which we are running. Here is the statement which we can run to stop the trace. We can run this final statement and delete the trace from our system. That's it. We are done with our primer of SQL Profiler and SQL traces. In the next clip, we'll start talking about the exciting topic of extended events.

What Are Extended Events?

So far in this module, we have learned about SQL Profiler and SQL traces. Now we will learn about extended events. Extended events are a lightweight monitoring tool which collects data about inner operations of SQL Server and helps troubleshoot performance problems. As a SQL Server performance tuning expert, I'm a big fan of extended events. There are three primary reasons I like to configure extended events over any other tool for analyzing SQL Server execution plans. Let's see these three reasons. Reason number one, they are lightweight on resources. Compared to any other solution, extended events are very lightweight and they add very negligible overheads.

Reason number two, they are very extensive and comprehensive. They pretty much cover every single SQL Server events and they are very accurate in capturing them as well. And finally, they are portable. I personally have created a few extended events and keep them very handy with me. If I see my client has problems with the CPU, I will deploy CPU-related extended events. If I know they have runaway queries, I immediately find extended events related to runaway queries and deploy on their server. Pretty much we can write one-time extended events and deploy at any number of customer place and we can also configure them very easily. If you have to create SQL traces, you have to use SQL Profiler; however, extended events are pretty standalone and extremely portable. These are the three primary reasons I am a big fan of extended events. Why don't we jump to a demonstration and understand extended events in depth in the next clip?

Demo: Extended Events

So far we have learned the theory about extended events. Now we will see how we can use extended events to capture our query plan. This is a very efficient method and I use it all the time in my SQL Server performance-tuning projects. So let's go to SQL Server Management Studio and learn how we can use extended events. Here we are back into SQL Server Management Studio and on the left side we can notice Object Explorer. Expand the folder Management and the fourth option is Extended Events. Further expand this one, and now expand sessions. Here are all the sessions which are available for extended events in our SQL Server. The one with the red dot on the top are disabled extended events sessions and the one with the green on the top are enabled extended event sessions. Let's right-click on Sessions over here and it will bring us to options, New Session Wizard, and New Session. We will use New Session option in our demonstration. If you want a very simplified version, you can also consider using New Session Wizard as well. Now click over here in New Session and it will bring up this screen. Over here give a session name as SQLAuthorityEE and now look at the option about scheduling this new session. You can select this option to start the event session at server startup. I will select the next two options because I want to start the event session immediately after session created and we will also watch the live data on the screen as it is captured. Now I will go to option of Events. This is one of the most important screens. Here we have to select the action name which we want to select. Look at how many different events are available in SQL Server. It is always a nightmare to select the right event to capture for what we want to do. This is where I use the option of category. This helps me to filter the data which I want to filter and select only the option which I want to follow. If I want data related to optimization, I can just select this and it will bring me only those events which are related to optimization. Let's go and select once again all the options. Now we want to capture the plan. Let's try word plan in our filter. There we go, we have so many different events related to plan. Now to see the plan of any query, we have to select the option query_post_execution_showplan. Let's select this particular option and click on this arrow so it will add that event to our selection. Next, click on Configure and it will additionally display all the global fields which we can include, along with our event. I always like to know database name, username, plan handle, query hash, and a session ID. Now if

you also want to see the text of SQL, you can further select this option. Next, go back by selecting over here and click on Data Storage. On this screen we have all the targets which are empty. That means our data is not stored anywhere. Let's click over here and select our very first target where we want to store our data. I like to store my data into files; hence, I will select event files, and over here give the name of the files. The filename is automatically provided, so I will just give part for my data storage. You can leave the maximum file size to 1 GB and click on enable file rollover. This means when your file is 1 GB big, SQL Server will automatically create new files and start recording all the events into that file. Maximum number of files I have given to 5 because I usually take backup of the file as soon as they start filling up. This is very, very critical because when I go for performance tuning consultation, I configure this particular event on my client's server and watch how the server is running various queries and capture their plan. Later on I can put various different filters on the top of those plans and identify poor performing operators and further tune them. But that's a conversation for the later module. Right now let's capture all the plans and later on we will further analyze them. Now you can go to Advanced option, or just directly click on OK. I will leave all this Advanced option as it is and click on OK. On the left side, you can notice my extended event is automatically created. This has also started to capture some data. As the database name is master, I can tell you that it is capturing some query which is ran by SQL Server itself in Master database. Now we will go to our file, which is Workload file, and run the first three queries by clicking on Execute. Let's go back to our live data and very soon we will notice various results over here. Last three lines are just added. The database name is master because when I ran this query, I left the database as master. Over here, the context of database was WideWorldImporters, but SQL Server does not look at this one when it is recording extended events. Let's change this one to WideWorldImporters and now run these three queries. You will quickly see the difference in our live capture data because the last three rows will now start showing that particular database in our event details. If you want to see what is the query plan it has captured, you can scroll down all the way and observe various columns. The object_id where the query was run was this and query was Dynamic ADHOC query. Here is the plan handle and query hash. Further scrolling down, we also have a column of ShowPlanXML. This is the XML of our execution plan and we can save that and open them into our SQL Server Management Studio. The very last line is SQL text of our query. Over here it is important for us to notice that we also have additional tab of Query Plan. Click on this tab and here we go, this is the execution plan of our query, which we can see in our extended events. Right now I can notice that this query is doing constant scan of 92% and that is scalar operation. Is it bad or good? Well, that's a conversation for later on. Right now this is a very, very efficient way to see what's going on inside your query. Let's get back to Details and click on column nt_username. If you want any of the columns from your details into also display, you can just click on Show Columns in Table, and now you can also see that particular column over here. This is very helpful because if I want to see execution plan of the query, which is returning us the largest amount of the rows, we can easily spot them in our table over here. For example, I may not spend time looking at execution plan of this query because it has only one row; whereas, for the selected query, we have 97, 000 rows and that might be my troublemaking query. Click on Query Plan and you can further dive deeper into various operators. As I'm storing all these extended events into my

file, I can go back and open this event file at any point of time and read details about my execution plans. Well, this is how you can use extended events to capture your query plan and further analyze them. And finally, if you want to put this particular extended event to any server, you can just right-click on the top of it and script this extended event to a new window. Here is our script for this extended event session. You can copy this and run it on any other server and immediately start capturing various relevant data. Once you get used to extended events, you will find Profiler and query traces not as helpful. Now let's summarize our entire session in the next clip.

Summary

In the previous clip, we have seen how extended events work. In one line if I had to summarize this module, I will say extended events are the most efficient way to analyze query plans for SQL Server. Well, with this we conclude our discussion about extended events and how they help analyzing query plans. In the next module, we'll discuss about how to identify poorly performing query plan operators.

Identify Poorly Performing Query Plan Operators

Introduction

Hi, this is Pinal Dave, and I welcome to this module where we will discuss how to identify poorly performing query plan operators. In the previous module, we have discussed how we can use extended events to catch query plans, and now in this module, we will understand how we identify who are the troublemaker operators in query plan and what would be the solution. Let's quickly see the agenda of this module. First, we will discuss about query execution plan and right after that we will discuss how to interpret query execution plan. Following that, we will understand about slow performing operators and also understand how to find root cause of them and a resolution. Theory of this subject will take us to certain level, but to understand query execution plan, demonstrations are very important and we will see three different demonstrations about poorly performing query plan operators and their resolution. Let's jump to next clip and start discussing query execution plan.

What Is a Query Execution Plan?

George Patton said, "A good plan, violently executed now is better than a perfect plan tomorrow. " As a matter of fact, if you know SQL Server engine, it definitely follows this particular advice. SQL Server engine always looks for a good plan and as soon as it finds one, it will immediately execute that. If SQL Server engine spends time to identify the perfect plan, it's quite possible to just find that plan. It will take quite a long time, and that would be not desirable by end users. Well, now you know the basic philosophy between SQL Server engine plan, you can clearly understand how come SQL Server sometimes have poorly performing operators in their execution plan. This is because instead of looking for the perfect plan, SQL Server presents query plan when it finds just good enough plan. So, what is a query execution plan? Well, a query execution plan is a visual representation of the operations performed by SQL Server engine to return the valid data for the query. When it is looking for query plan, it always looks for best possible option, but sometimes it stops a bit early and gives you just a good enough plan. This is why we need to identify poorly performing query plans and tune them ourselves if you think the query is performing not as per our expectation. Now let's understand how we can interpret query execution plan. First of all, query execution plan is read it from right-to-left. Most of the language in the world is read from left-to-right, but execution plan is preferred to read from right-to-left, but if you read your execution plan from left-to-right, it doesn't make you any different from the person who is looking from right-to-left. My preferred way, or many other experts' preferred way, is right-to-left. The reason why we try to read execution plan from right-to-left is because the operator with the highest cost is usually

located on the right side. This helps us to identify who is the troublemaker or which operator needs our attention first immediately. However, it's quite possible sometimes the most expensive operator is also located on the left side and we have to pay attention to that once we encountered it. If what I just mentioned is a bit confusing, don't worry; when we see a demonstration a little later in this module, everything will be very clear. Additionally, query execution plan helps us to understand data flow. By looking at various operators, we can understand how much data each of the operator is possessing and how many times it's being executed. Query execution plans are also very helpful with the warnings. We can see a plethora of different kinds of warnings when we look at execution plans. For example, if there is a memory pressure or TempDb pressure issue, query execution plan will immediately warn you with the help of operators. I find query execution plans very helpful when I'm looking for low-hanging fruit to tune my query. Also, query execution plans give us important suggestions and hints. For example, it tells us what kind of index can help your query to run faster. Yes, it is definitely true. SQL Server quite often knows that what kind of index you can create on your system to get more performance. We will discuss about this in depth a little bit later on. When I'm looking at query execution plan, I always pay attention to query cost in batch. Well, query execution plan is very important, but I'm going to add one more very practical real-world tip for you. That is Statistics IO and Statistics TIME. These are definitely part of query execution plan, but they are not visible immediately. You have to write a special command and you can see them into messages window. What we will do next is look at a very simple demonstration of query execution plan and we'll discuss each of these points in detail with the help of SQL Server Management Studio and a simple query plan.

Demo: Interpreting Execution Plan

Let's see our very first demonstration where we will open an execution plan and understand how to interpret it. Let's go to SQL Server Management Studio. Here we are in SQL Server Management Studio and I have written a very simple script based on simple database WideWorldImporters. Here is the script. Let's run this script, but before we run it, let's enable execution plan. Click over here and it will include Actual Execution Plan for this query. Now click on Execute. Once the query is completed, it will show an additional Execution Plan tab on the right side. The Execution Plan tab is now visible. Click over here and execution plan is here. But right now on the top part, there is a query, and on the bottom part there is an execution plan. You can change this slider in between and increase or decrease either of the section. Particularly, this is good if you're writing query and continuously seeing the execution plan. However, personally I do not like this setting, so I'm going to switch what I use at my client's place. I like to see my execution plan in the full screen mode. That way I can have more real estate and clear understanding of what each of my operators are doing. For that, I go to Tools, click on Options, and over here go to the settings of Query Results where I select Results to Grid, and now enable these two checkboxes for Display results in a separate tab, as well as to Switch to results tab after the query execution. Click OK and now you are back on this screen. Close any of your existing query and now reopen them. This time when you execute this query, you will notice that query results are

displayed in the larger area and execution plan, once query is completed, will be also visible in the next tab of execution plan. Now you can see one more option you can exercise is to right-click in the empty space and select Zoom to Fit. When you do Zoom to Fit, SQL Server Management Studio fits the entire execution plan in the single view for you. Now we are ready to interpret execution plan. As I mentioned, I like to see execution plan from right-to-left. Over here we have a lot of information, but the first thing which we must notice is number of rows which we have. Currently I have number of rows are 156, 016 rows, so this is the final number of the rows which we are returning in Results section. Now let's go back to execution plan, and on the right side, we can see we have operator of Clustered Index Scan. As soon as I mouse over this operator, SQL Server popped up yellow tool tip and displayed us quite a lot of information. The one on the top is Clustered Index Scan, and that is the most important thing to remember. The cost of this particular operator is 20% and it's visible on the screen, as well as when you click on tool tip over here, you can see under Estimated Operator Cost. Remember, our total number of rows is around 156, 000, whereas this particular operator is right now returning 48, 194 rows. Is it good or is it bad? Well, that's not something we can decide at this point of time. Only thing we can say that this operator is important and it takes around 20% of the cost. Next, we have Clustered Index Seek over here, and this time we can also notice that cost is 23% and it is also returning around 48, 000 rows. Remember, this particular operation is performed on different tables. Look over here under Object section. There you will notice the name of the table which is Sales.Orders. Now we will go to a next operator on the very right side, which is over here. This is also Clustered Index Scan, and it is on the table Sales_InvoiceLines. This particular table is now reading around 228, 000 rows, and that's a lot more rows than what we are retrieving in our query. From this particular execution plan, we can understand that each of the operators is responsible for a different number of rows and together eventually they will provide us only 156, 000 rows. Now, let's start going left further in our execution plan. When our query has executed, our invoices tables and orders tables together help produce 48, 198 rows, and both of them are joined over here under this operator in Nested Loops. The physical operation is Nested Loops, but logical operation is Inner Join. Remember, physical operation is the operation which happens in your storage system and logical operation is the operation which you indicate your SQL Server to do that with the help of query execution. Together, both of these tables are still producing around 48, 194 rows. Now, when we come over here to InvoiceLines, this is producing quite a lot more rows than what we needed. Right left to this particular operator of Clustered Index Scan, we have another operator which is Sort. The cost of this operator is around 48%. Let's click over here, and now it demonstrates that this operation is still producing the same amount of the rows as many were provided to it. Sort was produced because somehow our results set requires an Order By condition. Please remember when I wrote this particular query, we did not write anything to order our data; however, when our query was provided to SQL Server engine, SQL Server engine internally decided that it needs to do a merge join, which is more efficient in this scenario, and for merge join it needed all the data from this particular table, and to provide all the data, it has to implement a short operation. Even though SQL Server Management Studio has to use the Sort operation, we can observe in our case that this is very expensive. First of all, it is taking around 48% of the cost and also there is a warning. Please pay attention to this particular line where it says operator

used tempdb to spill data during execution with some more information. As a matter of fact, this sort operation is what is actually slowing down this entire query. And when you see this operator closely, there is a small triangle or yellow bang on the top of it. That indicates there is a warning, and with the help of mouse over, we can see the warning and find an appropriate solution for it. There are a lot more details which we need to understand in terms of execution plan; however, our goal for this demonstration is to give you a brief overview about how you can quickly interpret your execution plan. Next, we will understand some slow performing operators and how we can optimize them.

Slow Performing Operators

Slow performing operators, in query execution we always have to look for an operator which is creating trouble for us or slowing down the entire query. It is quite possible that we may have a workaround for that particular operator or we can lose something behind the scenes so that operator speeds up. When I go for any SQL Server performance tuning consultation and I have to tune query, I look for slow performing operators. Let's discuss three of the real-world scenarios which I often encounter at my client's place. Number one, slow performing operator is scan. Please note that scan is not necessarily always bad; however, in my experience, there is always some workaround to speed up. Some people think seek is faster than scan. That is true, but sometimes scan is faster than seek as well. Well, we are not here to discuss about scan versus seek, but we will see later on in demonstration how scan can be converted to seek and gain additional performance. Similarly, parallelism is another interesting topic. A lot of people think that when query runs on multiple CPU or uses multiple threads, it runs faster. In my experience, I have seen queries which have to use parallelism can be always tuned and made run on single CPU. We will see a real-world example of this slow performing operator as well a little bit later on. And finally, I always look for implicit conversion in any of my query plans. If I see implicit conversation, that means my query plan is not able to efficiently use my indexes, and that is a big concern for me. I am not a big fan of creating indexes; however, if I already created an index, I want my queries to use that, and implicit conversion is one of the slow performing operators or a troublemaker which prevents that. Now we have discussed about these three slow performing operators, let's go and see a real-world demonstration and understand how each of these works.

Demo: Performing Operators - Scan

In this clip, we will see demonstration of how we can find poorly performing query plan operators and tune them. First, we will start with scan, and in the next clip, we will look at parallelism. After that, we will look at implicit conversion as well. So let's start with our very first demonstration about slow performing operators about scan. For that, we will go to SQL Server Management Studio. Now here we are in SQL Server Management Studio, and we will be using WideWorldImporters sample database. First things first, we will enable STATISTICS IO and TIME ON. As I

mentioned to you earlier that along with the execution plan, it is very critical that we also keep on checking STATISTICS IO and TIME; otherwise, we will not know if our query plan is improved or it is still giving us the same old performance. Now let's execute this statement. Next, we will go to our query and run it. However, before we run this query, let's enable the Actual Execution Plan. Next, execute the query and execution plan is here next to Messages. Click on Execution plan and here in this query we can notice that this is a huge execution plan. Remember, our technique is to interpret query execution plan. We will go all the way to the right side and start looking at if we can find our expensive operators. So far, what we have seen, our query operators are not expensive. Scroll down a bit more and here also we do not see any query cost very high. When we scroll a little bit on the left, here is an operator which catches our eye. Cost of this particular operator, which is highlighted, is 67%. When you look at other operators' costs, like 5%, 0%, 2%, 67% is definitely a lot more cost than other operators. We can definitely start tuning this particular query plan from this operator. Remember, it's not always possible that we can find some kind of workaround or tuning solution for slow performing operators. Despite that, it's our responsibility as performance tuning experts to look for what we can do to tune this particular operator. Now let's mouse over this operator and see various properties. The very first thing which attracts me is Clustered Index Scan. The table used here is InvoiceLines. That means SQL Server has to loop Clustered Index Scan on InvoiceLines table and that seems to be a very expensive operation. Now, to double-check that, let's go to our Messages tab, and here we can see statistics of our queries. Our query is taking around 827 ms in terms of CPU and total time around 2936 ms. This is almost like 3 s. Let's run this query one more time and go to Messages. This will remove any doubt in our mind if the query was reading data from cache or not. Now here is our updated time, which is very similar to what we have seen before. When we go up over here in this highlighted line, we can notice that InvoiceLines table is getting scanned and total logical reads are from this table around 5160 pages. Each of the pages in SQL Server is of size 8k. That means SQL Server has to read around 5160 pages multiplied by 8k equal to 41, 280 KB. Let's divide that by 1024. That's a size of a megabyte in front of us. That means to retrieve this query with 156, 000 rows, SQL Server has to read around 40 MB of data from InvoiceLines. Similar mathematics we can also do for Invoices table as well. Now our responsibility is to tune our query and we have already noticed that there is a scan operation on InvoiceLines which is very expensive. One of the very first thoughts that comes to my mind is to create an index on this query. Now I can go and spend time to figure it out what kind of index I need or I can also consider looking at a query hint, which SQL Server provides us over here in our execution plan. Please note that always take advice from SQL Server; however, do what you think is best for your query. Here when I mouse over and right-click, I can see various options related to this execution plan. Let's click on Missing Index Details. Here is the script which is generated by SQL Server Management Studio, and this is the missing index which SQL Server thinks I need to create. Please note that I have scan on InvoiceLines table; however, in this script, SQL Server is creating index on Invoices table. This is a very interesting point where I have to decide, should I spend time on InvoiceLines table or just believe what SQL Server is suggesting? As this is not a SQL Server performance tuning course, but rather a course about analyzing query plans, we will go with our SQL Server recommendation. I will copy this script and

create this particular index for our query. Let's go back to our script where we were before and here is the index I have built from the suggestion which SQL Server has provided. Let's create this index, and now we will go back to our query one more time. Let's execute the same query and now we will inspect execution plan one more time. Look at that. Our execution plan is absolutely changed. Previously we had a lot of parallel operations. All of them are gone, and our execution plan is much simpler than before. As we have tuned this particular operator which was not so expensive, SQL Server still sees a lot of improvement. However, our Clustered Index Scan on our InvoicesLines remains as it is, and as a matter of fact, the cost of this operator has even further gone up to 84%. As the cost of every operator is relevant to another operator, if one of the other operators improves its cost, it is natural to see a bump in the cost of other operators. Now, let's go to our Messages tab and see what our statistics suggest. Look at this table, we can definitely see improvement for our Invoices table, but we do not see much of the improvement in our InvoicesLines. The logical read remains still 5003, which is better than before, but still not what we want as our end result. Looking at CPU time and elapsed time, we can definitely say that this query is run faster than before. This is the point in time where we have to make a decision. Should we further tune this query or leave it as it is? Well, I'm making the decision to tune this query a little bit further. Let's go back to our query editor and now create one more index on index lines table. When I look at this query, I can see this query is joining on InvoiceID for InvoiceLines. Additionally, there is a column Description also retrieved in a select statement. I will put both of them in a single index and we'll create this index on InvoiceLines table. When I execute this query, it created the index. Now let's go back and run our query one more time. This time our query took a little bit of time, but we cannot say how much time it took unless we look at Messages and Execution plan. First step is to go to Execution plan. Further looking over here, we can notice that SQL Server has definitely created a very different execution plan and Index Scan has reduced a little bit over here. To our surprise, there is one more operator also introduced on Index Line table, and that is Index Seek. Let's go to Messages and see our query statistics says. Well, we can clearly see that index has improved logical reads, which was earlier 5000 and now it is below 3000. CPU time and elapsed time is also improved a bit. Definitely our newly created index was helpful, and we were able to reduce a lot of reads. Now let's go back to our execution plan. Further looking at our execution plan, we still have our Index Scan on InvoiceLines around 76%. That means if I have to further tune this query, I would once again focus on this table and see what different things I can do on this operator to tune it in such a way that it does much lesser reads than 3000 logical reads for this query. This is how you can take any execution plan and further analyze the system. For the purpose of this demonstration, we will stop here, and now in the next demonstration, we will see how we can tune poorly performing query plan operator, Sort.

Demo: Performing Operators - Parallelism

Now we know how to analyze our query plans, we will move a little bit faster in looking for the root cause of slow-running query and its resolution. In this demonstration, we will talk about the very expensive operator, Sort. If you

search for poorly performing query plan operators on the internet, you will definitely see Sort in the top three, and that is why we are going to discuss that now. Here is our demonstration for Sort, but before we continue that, we will go to our previous demonstration of scan and drop the indexes which we had created. Now let's go back to our demonstration of Sort and first things first, we enable the execution plan and enable STATISTICS IO and TIME for our query. Let's run our query and see its execution plan. Once query is done, click on Execution plan tab and here we have our execution plan. Definitely we can start focusing on our table scans, but that is what we have already discussed in previous demonstration. Now let's find if we see a sort operator in our query or not. Here it is. Sort operator is definitely there in our query plan. The cost of this operator is only 5%. That tells us that it doesn't look like a very expensive operator. Compared to that, I can clearly see the operator on the left side, which is parallelism, is expensive. This may come as a little bit of a surprise to many of you. We started this demonstration talking about sort is an expensive operator; now we suddenly are talking about parallelism. I really want to demonstrate to you that SQL Server query plan analysis is not as simple and straightforward as you like to believe. Every single day I get lots and lots of email where people are asking me can you teach me how to get rid of sort operator from our query plan analysis? The answer is very simple. Sometimes you can get rid of the sort operator, but sometimes it is just there to stay because either you use STOP, ORDER BY, DISTINCT, GROUP BY, HAVING, or some other operator which for sure require ordering of your data, and this particular operator may be very useful. When you encounter such a situation where sort operator is not as high as you'd think, you must pay attention to its neighbors and start looking at how to tune them. In this particular case, the parallelism seems to be an issue right now. And truth be told, in the year 2000 I also believe the sort is very expensive, but as time passed by, ecosystem of SQL Server changed, I started to believe that sort is just there to stay and it is not as complicated or expensive as I believed. As a matter of fact, in the latest version of the SQL Server, I find parallelism has taken the place of sort operator and whenever I see my query going parallel, I am extremely alert. Any query has to go only parallel because query is extremely expensive. That means we need to look for the expensive operator and see if we can tune that. Over here when you mouse over this particular query hint it is only talking about creating index on Invoice Table, which we have already seen in previous demonstration. This definitely helps our cause, but that's not what we want to do in this demonstration. Let's go back to our query and try to think what can we do to remove parallelism from this query. Will that fix this particular query's performance? This is the question which we want to answer. Remember, what I am trying to demonstrate to you, many of you will not agree immediately unless you run this demonstration or similar demonstration on your machine or dev production systems. A lot of us think that when we have more CPU or more threads to run, our query runs faster. That is not always true. As a matter of fact, on real-world scenario, I always find that's not the case. Now look at over here, we have InvoiceLines table, which is doing is logical reads around 5200 rows. The Invoices Table is also doing logical reads of 1389 rows. Now click on query editor and let's think what can we do so query does not go parallel. There are two things which we can do, but the first thing which comes to my mind is that I'm going to restrict this particular query to run on only one degree of parallelism. Currently when you go to execution plan and click over here, you will notice that there is number of execution which says 8. That means this query is running actually on 8

processors or 8 threads. Now we know that our query is using 8 threads, now let's go back and provide option max degree of parallelism 1. That means no query will use only 1 CPU. Let's select the statement and click on Execute. Upon execution, we have new execution plan where there is absolutely no parallelism at all, but the question which comes to our mind is that if this query is faster or not. To validate that, let's go to Messages, and here we can see various details related to our query. Definitely page reads from InvoiceLines has reduced to 5003. That means this table is now reading around 200 less pages. The biggest improvement which we can notice is in Invoices Table where logical reads are reduced from 1200 to just 500. This is a huge improvement in our case and query time had improved for sure. This was the step number 1 to improve performance of our query. The next step is either we can create index which is suggested here or can further look for expensive operator like Clustered Index Scan on table InvoiceLines and create index on this table. Let's do that task as well, and we will see final performance once the index is created. Now let's execute this query and see what it has to bring us to the table. The execution plan is still very similar, but index scan has reduced from over 80% to just 38%. Now click on Messages, and here it is. We can see a lot of improvement in terms of InvoiceLines because from 5000 now the logical reads are reduced to just 3000. This is how you do a performance tuning in real world. Remember when you see a query, just don't blindly put OPTION max degree equal to 1; there are a lot of things you may want to consider. What we have seen in this demonstration is just the tip of the iceberg. Personally, I'm very much against providing query hints. I like to rewrite queries in such a way that I do not need to provide query hints. However, that topic is not in the scope of this course, so we will move to our next demonstration where we will discuss about implicit conversion. Right before we go there, let's drop the index which we have just created by executing this script.

Demo: Performing Operators - Implicit Conversion

Now let's look at the final demonstration of this module. First things first, we will enable the execution plan and enable STATISTICS IO and TIME. Next, we will run this simple query, which is retrieving data based on CustomerPurchaseOrderNumber. I have selected a customer purchase number over here and I'm executing this query. Upon execution of the query, when we check the execution plan, we will see again parallelism over here. Definitely we can improve this parallelism by providing max degree hint, but as I mentioned to you earlier, I am not a big fan of providing query hints. Let's see what else we can do to reduce our overall cost of this particular query. When I look at our SELECT statement on all the way to the left, there is a small triangle on the top of it. Let's see what it has to say in warnings. The warning says there is a type conversion in expression. There is a column which is CustomerPurchaseOrderNumber is converted to integer, and that conversion is affecting cardinality estimation in query plan, and that leads us to a unique scenario where we may not get seek in our query. While that's pretty interesting because I thought CustomerPurchaseOrderNumber would be an integer column to begin with; hence, I was comparing an integer. But from what I'm looking at right now, the warning says definitely that column is not integer and it has to be converted to integer. That is, that might be the reason why we are seeing Clustered Index

Scan on table invoices over here. Let's go back to our editor and mouse over the column CustomerPurchaseOrderNumber. A mouse over clearly indicates that this is nvarchar column. This is indeed interesting, or maybe just a mistake of database architect. However, at this point in time, we cannot blame him for what he had done in the past. Let's go and fix our query in such a way that we can get optimal performance from it. As data type of this particular column was nvarchar, let's go back and change data type of our comparison from integer to varchar. Now this is a string which I'm comparing to nvarchar column. Let's execute this script and see what changes we might have in our execution plan. Now click over here and here we go. Suddenly we see a missing index hint, which can improve our query's performance by 99%. There is also no warning on our SELECT. Whereas when we click on this operator, it still says Clustered Index Scan, and that scan is 99%. Let's go back and run both of these queries together and compare if there is any improvement in bottom query from the query on the top. When I execute both of these queries together, it clearly says that query performance of both of these queries is absolutely the same. However, as we have used the correct data type in the comparison in bottom query, we can get an additional hint of missing index. Let's try out by creating this missing index on our query. I have already typed up the script of the missing index hint. Let's execute this script and now let's go back to our queries. Select both of these queries and click on Execute. Queries are executed, and before we see execution plan, let me ask you a question. What do you think, which one of these queries will perform better? Query on the top, or query on the bottom? Please write it down on a piece of paper so when you see the answer you have no option to change what you just guessed. Now let's go back to our execution plan, and over here it is very clear that query on the top is not performing as well with the help of index, but query on the bottom is doing amazing with the index which we have created. That we can prove by looking at query cost relative to batch. The query cost relative to batch for the query on the top is 92%, which is very, very high; whereas query cost relative to entire batch for bottom query is just 8%. That means query on the bottom is just taking 8% of the resources where query on the top is taking around 92% of the resources when both of them are run together. A query is performing better when the query cost relative to batch is a lower number. To validate what we just discussed, let's go to Messages, and over here we can compare the logical reads of both of these queries. The query on the top is doing a huge number of logical reads from invoices tables. It is around 369, whereas query on the bottom is just reading 26 pages. On the top query, we have two additional tables created, which are Workfile and Worktable. They both indicate the top query needs help from TempDB. In the bottom query, those two tables do not exist. That means the query ran successfully without help of TempDB, and, hence, it is performing much faster. Query execution plan is also very visible. Query on the top takes 32 ms CPU time, where query on the bottom, the CPU time is next to nothing. Well, there you go. You clearly learned today that with the help of indexes or doing a little bit of rewriting, you can improve your query plan. When I go to performance tuning consultancy, I always look for the most expensive query plan operator and see what I can do to reduce its cost. Now that we have seen three important demonstrations of poorly performing query plan operators, let's go to summary.

Summary

In the previous clip, we have learned about how to identify poorly performing query plan operators and its resolution. Here is the summary of this module. Always check query execution plan before deploying queries on a production environment. You may find one or two operators which are very expensive, and by just doing a little bit of tweaking and tuning, you can improve performance of your query. The best way to overcome poorly performing query plan operators is to rewrite queries and use efficient indexes. We have learned about them in detail with three important demonstrations. Now in the next module, we will talk about the very important concept about Query Store and how we can create efficient query plans with the help of this new feature.

Create Efficient Query Plans Using Query Store

Introduction

Hi, this is Pinal Dave, and I welcome all of you to new module Creating Efficient Query Plans Using Query Store. Query store is introduced in SQL Server 2016 and it is quite popular among SQL Server performance tuning experts. They often call it as black box for SQL Server because it captures history of executed queries and various statistics related to it. We can always go back to query store and ask information about SQL Server past and make better decisions for the future. Let's see the agenda of what we are going to cover in this module. First, we will have a quick introduction about Query Store, and right after that, we will see a demonstration related to setup and configuration of query store. Following that, we will also talk about various reports of query store. These reports are about real-world scenarios and performance problems. To understand query store in depth, we will see some really interesting demonstrations. Well, that's the agenda for this module. Let's start with a quick introduction of query store in the next clip.

What Is a Query Store?

Zig Ziglar said, "Unless you have a definite, precise, clearly set goals, you are not going to realize the maximum potential that lies within you. " This is exactly what query store does. Query store helps us to understand past of our query plans and helps us build better execution plans based on various statistics and information. Query store is a very powerful tool put in the hands of SQL Server performance tuning expert. Let's talk a little bit more about query store. What is a query store? Here is the official definition about it. A query store tracks query plans, statistics, and historical data to help diagnose performance degradation of the queries over time. Remember the key word is over time, and that's exactly what we are going to see in the demonstration a little bit later on in this module. Query store is database level feature and it is available in all editions. You don't have to worry that if you have a standard edition or enterprise edition. Query store will be available with you if you are using SQL Server 2016 and later versions of SQL Server. It's also becoming a primary tool to find performance regression. I use it quite often to identify resource intensive queries during my real-world performance tuning engagements. Sometimes when a client complains that their query is no longer running as efficient as it used to run before, I depend on a query store to help me out with execution history and also understand what query plan was built at that point in time. I can always use the earliest successful efficient execution plan and force it on future queries to get optimal performance from SQL Server. Yes, what you heard is absolutely true. SQL Server can now use its own history to tune its future. This is very, very

powerful, but to see that particular demonstration, first we have to configure our query store. Let's see in the next demonstration how we can get started with the query store from scratch.

Demo: Query Store Setup & Configuration

Query store is one of the most popular features released in the recent version of the SQL Server. I use it quite often at my client's place when I am doing real-world performance tuning consultation. Instead of setting up some expensive monitoring tool, I use query store to record the history of SQL Server execution plans. Let's see in this demonstration how we can set up and configure query store. Now we will open SQL Server Management Studio. Here we are in SQL Server Management Studio and on the left side we have Object Explorer. First, right-click on the database on which you want to configure query store. Go to Properties, and on this screen, select the option Query Store, which is last on the left side. On this page, we can configure query store. Currently, the operation mode is set as off. That means SQL Server's query store is turned off. Let's see what are the other options which we have. When we click here, the very first option says Read Only. This mode suggests that new query statistics will not be tracked, but we can always go and check history of the earlier query plans. The second option is Read Write. This is what we are going to select for this demonstration, as it will allow us to capture query execution plans and its relevant statistics. As soon as I select that, you will notice that various configuration options are enabled. The Data Flush Interval is a very interesting setting. The default is usually set to 15; however, I have changed this 1 to 1 for this class. I personally believe Data Flush Interval at value 15 is just all right. What this setting does is at this particular interval, query store will collect various query plans and runtime statistics from the memory. If you set this one to very aggressive setting as 1, then query store will collect it more often and you may start seeing your data in various reports quite real time. When I'm doing real-time performance tuning, I set Data Flush Interval to 1 minute, but if I'm collecting the data to analyze later on, I will leave it as 15. Similarly, Statistics Collection is currently set to 1 minute as well, and you can change it to any other minute over here. It is very interesting to see how SQL Server Management Studio works. The first interval is definitely set in a minute, and the minutes are written in the left side, but for statistics we can set duration in even days, and that's why the unit is given on the right side in our drop-down. Remember, by default, this value is also set to 60 minutes, which is 1 hour; however, for this demonstration, I will also change to just 1 minute. Data of query store is stored in the database itself. That means whatever the max size you keep it over here, that's what amount of data will be stored in the query store in your database. In other words, your database will grow by this much size if the query store is completely full. This is very important because if you take your database from this server or instance and install it somewhere else, you will be able to see all the history of your query execution and statistics on another server is the query store data is stored in the same database. Query Store Capture Mode I have left it at Auto and Size Based Cleanup Mode I have also left it as Auto. If you want to capture all the queries, you can change this one to All, but if you leave it as Auto, SQL Server will capture selected queries based on the resources available. On another note, I like to keep Size Based Cleanup Mode to Auto. If I turn

it off, then I have to manage query store myself and I totally believe in automation. Finally, here is the configuration which indicates the duration to retain query store runtime statistics. Currently it is set to 30 days. That means query store will keep your data for 30 days and after that it will automatically remove. I think 30 days of history is good enough if you want to analyze any database performance. And finally, we have two charts over here. On the left side, we have pie chart for WorldWideImporter database and how much portion of it is of query store. As we are just configuring query store currently, it is 0 MB. Similarly, on the right side, we also have query store pie chart. This indicates that query store can grow up to 500 MB and currently the data or statistic which is stored in this 500 MB is 0 MB. This is because we are just configuring query store. We will come back to this setting a little bit later on and notice the data in this pie charts would be different. If any point of time you want to remove all the data related to query store from your database, you can just click on Purge Query Data and SQL Server will remove any data and statistics related to query history from your query store. Use this sparingly because this is an irreversible action. Now click OK, and this will configure your query store for database WideWorldImporters. Now to validate that we have successfully configured query store, we will run this query. SELECT name and is_query_store_on from sys.databases. Let's execute this query. Now we can confirm that our database WideWorldImporters has query store feature enabled. That means if we start running any query on this database, that would be recorded in query store. We will see the demonstration of this a little bit later on in this module. Next, we will talk about three important query stores and query store reports.

3 Important Query Stores

In the previous clip, we have learned how we can configure query store, and now we will see three of the important query stores. Here are three important query stores in SQL Server. First is for plan store, second one is for runtime statistics, and third one is for wait statistic stores. All these three stores contains various data related to plan, statistics, and waits. To get this data in a meaningful way, we have to depend on query store reports. There are many query store reports available right out of the box in SQL Server. Let's talk about these query store reports which we can use to retrieve various information from our query stores. The very first report is about regressed queries. This is where we can go and see which queries which were performing better earlier now performing poor. The one after that is about overall resource consumption. This tells us how much CPU, I/O, and memory SQL Server is consuming at any particular point of time for any query. This is very, very essential if you have resource crunch on your system. The next one is top resource consuming queries. This report tells us which queries are consuming maximum CPU and I/O. If you ask me, I use this report quite often with my clients. Developers often ask me which of the queries which they should be tuning. I always point them to this report and tell them if they recognize any of this query, once they recognize this query, they need to look for poorly performing query plan operators and find solutions or workarounds to optimize those operators. We have learned about how to tune or optimize query plan operators in the previous module. You can use that knowledge once you know what is the plan for your top resource consuming

queries. The next report is queries with forced plans. This report will only contain data if you are going to force optimal plans for your queries. What essentially this means is that at any point in time if you find a query which was running optimal in the past and slow presently, you can take the plan from the past and force it to use it for the future. This is a very, very powerful feature of query store, and we will see that a little bit later on in this module. This report will contain all those queries which have forced plans. The next report is about queries with high variation. If the queries are taking high resources at one point in time and low resources very quickly, SQL Server will also log them in a query store and produce them in these results. Finally, my favorite report is about query wait statistics. I use SQL Server's wait statistics to understand what is overall performance bottleneck for your database. This is very powerful with query store as now we see query wait statistics along with poorly performing queries and its execution plan. Finally, there is a report for tracked queries. We can take any queries in SQL Server, track its performance at prescribed intervals.

Demo: Query Store Reports

In the previous clip, we have discussed about various kinds of query store reports. Now it's time we see them in action and also explore options to customize them. Let's go to SQL Server Management Studio. Here we are in SQL Server Management Studio and in front of us is a workload file. All the scripts which we have seen in previous modules I have put them together in this single file and we will run them now. This will create a workload for our database and all these queries will be recorded into query store. Remember, the Data Flush Interval is 1 minute; hence, after all these queries are run, you have to wait for minimum of 1 minute before you go to query store reports. You do not have to wait for all these queries to run and wait for 1 minute because I am going to edit out that waiting period once the queries are completed. Now we have waited for 60 seconds, let's go to query store and explore various reports available to us. First, expand the folder of Query Store. Right below there, there are seven different reports. Let's first click on Regressed Queries. I don't think that we will see any queries in these reports right now. This report is only for the queries which are slowed down from the previous run. As I have just run these queries a few seconds ago, they do not have any chance to slow down. However, the next report is very important for all of us. That is, overall resource consumption. Right-click on it and click on View Overall Resource Consumption. Here it is. There are various details related to overall consumption of resources. For example, right now there has been a total of 139 queries and the duration of all queries run is listed here. Similarly, CPU time, total logical reads and writes, along with physical writes are listed over here. Total max degree of parallelism is also listed in this column. Upon scrolling on the right side, there are various other important information; we can also see them together in this row. This particular report is about overall resource consumption on your system. At various intervals, you can open this report and see how much of the resource your SQL Server database is consuming. Quite a lot of times, a lot of users come to me and say that their server is busiest between 4 and 5. I always argue back saying how can you prove that? They don't have any answer to my question. If they were using query store, they can easily use this report and

tell us how exactly their resource is being consumed. Now let's go to the next report which says Top Resource Consuming Queries. This is a very, very critical report because from these reports developers or DBAs can find out which queries are taking maximum resources. Remember, top 25 resource consumption for database WorldWideImporters and Time period just last one hour. You can always configure this time and various details from this step of configure. Click over here and it displays various configuration options. For example, right now the default report consumption criteria is duration, but you can change it to any other properties by just selection the appropriate radio box. Also, the time interval says last hour, but you can customize it to last few days, minutes, months, or whatever time period you want. This is very, very powerful tool. I use it all the time at my client's place. The top 25 can be easily changed by changing this value. Now let's explore this report a little bit more. Currently this query is ordered by metric duration. When you select this particular option, it displays that there is a query which is SELECT* from sales invoice and it has taken maximum amount of time to execute. Various execution times have been listed over here. For example, this query, which you are seeing right now, has created query plan number 14, and execution time of this particular plan is different at different points of time. As I have put this query multiple times in our workload, it displays different execution time. This is very, very critical information. Remember, have you ever faced a situation where query was running fast and suddenly got slower? Do you know what was the plan when it ran fast and what was the plan when it ran slow? Well, you can easily figure that out if you have query store because it collects pretty much every information related to query execution. Now let's order this report based on CPU time. Here you go, we can find the same query which has taken maximum time for duration is also taking maximum CPU time. I have a hunch that I will see that once again if I also select Logical Reads. There you go, I am correct once again. Now you can also select any other query over here and find its relevant logical reads on the right side. Remember, logical reads for this query is going to remain the same, as there was no change of data; hence, you are noticing them all together at the same level. Further selecting wait times, it also indicates that the query which is currently selected has taken maximum amount of the wait for your query. This query can be sped up if we create this index which we are seeing in query hint. Well, this is just a guidance. As we have discussed in the previous module, you need to also be aware of the various parameters for your query. Well, that's good enough information about this report. The next report is about queries with forced plans, and as we have not forced any plans, we will not see anything here. Right after that, we have queries with high variation, which, once again, indicates the queries which have taken the highest amount of the variation in terms of the metrics which we have selected. Currently it is selected as duration, but you can change in terms of memory consumption and your queries and details will change. We have looked at wait statistics earlier, but we have seen queries which a large amount of the wait under our top resource consumption query, but we did not know what exactly wait they are creating. Well, that problem is resolved in this report. For example, highest amount of the wait is network IO, and when we select that particular wait, it lists all the queries which have run and generated that particular wait. For example, this query, which is SELECT* from sales customer transaction has created maximum network issue and this query is second one into that list. Let's click back over here and here we also have additional configuration options related to wait time and execution counts. For

example, the query which we are seeing on the screen has created maximum amount of network IO. Now go back over here and it will bring us to all other waits again. Select parallel waits and now we have all the queries we just ran in parallelism. For example, the query which you are seeing on the screen has ran on multiple CPUs and created a warning. We can look up that particular warning if you mouse over the SELECT statement, and it says there is CONVERT_IMPLICIT issue. We know that if we fix this CONVERT_IMPLICIT issue, maybe our query will start running on single CPU and this particular waiting situation of our queries will be removed. There you go. With the help of various reports, we can see our query's history and make informed decisions about its future. I use query store and its reports very extensively. The final report is about tracked queries, but as we are not tracking any queries right now, we will not see any details here. However, if you want to track any queries, you can type down query ID and click on go to see various queries. Well, with this, we finish learning about our various query store reports. Now we will go to our next demonstration where we will see a real world scenario about how you can use the information about query's history and tune its future.

Demo: Resolving Parameter Sniffing with Query Store

Let's see a real-world scenario. I have taken this demonstration from one of my customers and I have simulated it for you on sample database. I'm very sure that all of you have heard about parameter sniffing and I am also very confident that you know quite a bit of tricks to find workaround of that. However, it is equally true that no matter how much you try to avoid parameter sniffing, it always come back in one form or another form. Today we will see the real-world demonstration how we can resolve issue of parameter sniffing with query store in the demonstration. Let's assume at Time = A we are passing stored procedure Parameter A. At that time, we are getting A+, or an amazing performance. And after awhile, at Time = B, we are passing Parameter B to our stored procedure and that point in time we are getting performance D-, or very bad performance. Now the problem situation or problem statement in front of us is how to get optimal performance from a query at all the time with all the parameters. This is an interesting situation, and let's go and find a solution of it with the help of query store. Here we are in SQL Server Management Studio and we will see a real-world situation. This is the query which is counting total account person ID and on the right side we are also grouping that with AccountsPersonId. Now let's execute this query and see what it demonstrates. When you run this query, the top row demonstrates that with AccountsPersonID, 3260, there are 4 accounts and with AccountsPersonID 1001 has around 22316 accounts. That means there is a huge difference between both of this AccountsPersonID in terms of how many accounts they have. Now, before we continue, let's do one interesting task which we have not covered in previous demonstration. It is about clearing query store data. Right-click on your database and go to Properties. Over here select Query Store option and now both of these pie charts are displaying different information. For example, out of 500 MB available, query store has just occupied very little data. Let me just hit Purge Query Data so when we run our queries, we do not get confused with anything which we have ran earlier during our fake workload. Now click on OK and next we will create this stored procedure. This

stored procedure is getting accounts detail as per AccountPersonID. Now the stored procedure is created. Let's do a comparison between two of the parameters which we have identified earlier. Before we do that, we are going to clear the cache of our entire system. Please note that it is not recommended to run this on your production system because it will clear every single cache in your memory buffer, and you may see performance degradation. Now execute this, and it will remove procedure cache from my memory buffer. Enable STATISTICS IO and TIME and next thing is to run this stored procedure where we have small amount of data. Let's execute this stored procedure with enabling execution plan. Now when we go to messages, we can notice logical reads are around 14. That's a good number because we are only retrieving around four rows. Now when we go to execution plan, our execution plan demonstrates that there is a nested loop and key lookup. We just accept this as it is and go to our next query. This time we will be passing a different variable to our stored procedure and see what is our execution plan. When we execute this query, our execution plan is built also relatively quickly and we can notice the same two operators which we have discussed earlier. One of them is nested loop and second one is key lookup. Let's go to messages and look at total logical reads. The logical reads for this query is around 66, 990; that is around 67, 000. This is quite a lot of page reads compared to what we are retrieving; however, we can't make any decisions if this is good or bad because we do not have any other benchmarking data. Now go to query editor and clear the cache one more time. Next, we will run the same stored procedure. This time order of both of the stored procedures is reversed. For example, we will run query with huge amount of data first and see execution plan right after that. Upon clicking on execution plan, we discover a very different execution plan than before. We notice Clustered Index Scan, Parallelism, and an index hint which can improve performance of this query by 99%. This is quite different from what we have seen earlier. Let's go to messages and see how much logical reads this query does. This query currently does around 11, 994 logical reads. That means approximately around 12, 000 reads. This is way lesser data than previous execution. Now when I run the second query with small amount of data, let's see what our query plan messages shows us. Going to execution plan, we find exactly the same execution plan of Parallelism, Clustered Index Scan, and missing hint. Upon going to Messages, we see a big surprise because logical read is once again 12, 000 pages. This is quite huge compared to what we are getting in our query. Now is the time where we have to make a decision. It is very clear from this demonstration that our SQL Server takes plan of query which is executed first and saves it in buffer case. When the same query is executed again to a different parameter, it still uses the same execution plan. If you remember module two, we had discussed about this in the fourth step of query execution. This is indeed a troubling situation. What should we do so our query always gives us the best possible performance when we run them? Well, there is no right or wrong answer. That SQL Server performance tuning is always about trading. From one side we gain something and from the other side we have to lose something. Here if we are going to prioritize this query based on our business needs, we might get performance hit when we run this query. Now let's see how we can prioritize execution plan for this particular query. As a first step, let's go to our query store report, Top Resource Consuming Queries. Click on View Top Resource Consuming Queries. And now here, based on duration, we have multiple results. Let's change this metric to logical reads. Now for this particular query, we already know that there are two

different execution plans. In our case, query store has created two plans, one with plan ID 3 and another one with plan ID 4. You can change between both of these plans by just going over here and selecting relevant plan. This is the plan when we had run stored procedures with a huge account first and this is the plan when we had run our stored procedure with a small amount of data. Please note that we need to use always the plan which is optimal for our huge amount of data or huge account. Let's go and check what was the execution plan when our huge account was to be prioritized. Come back over here, clear the statistics, and run this stored procedure. If you notice, this stored procedure builds execution plan with parallelism. Let's go and prioritize this particular execution plan in our query store. Now come back to top results consumption query and select the execution plan where we see parallelism. Now click on Force Plan. That means every single time now this particular query runs, it will always use this plan. Click over here and it will ask you for confirmation. Click Yes, and that's it. Now you have just solved your biggest problem about parameter sniffing. Let's go back and validate that by running our stored procedures. Clear the cache first. Next, clear the cache first. Next, we run our stored procedure with small account or a little amount of data and see what execution plan it will build. It is very clear that it is building execution plan with parallelism. Also, logical reads are high, but we are aware of this situation. When we run our huge account, it's once again going to use execution plan with parallelism. No matter what we do, now our query will always use execution plan which is prioritized for our huge account. This is the advantage of our query store. There are many cases in the past when my customer says that my query runs faster with certain parameters or my query runs faster at certain times. I always look at top resource consumption query and see what are the different plans the query has created. Based on various different parameters, I made informed decisions to force plan or not. Please note that I could have always created this index and also got additional performance; however, that's not the feature I was covering in this demonstration. Well, I'm very sure that now you have a clear idea how query store works, and with the help of various statistics information about query's execution history, you can further optimize your future queries. Let's go to the next clip and summarize this module.

Summary

In the previous clip, we have learned how we can use query store to get maximize performance from our query execution plan. Let's summarize this entire module. Query store is an efficient and easy way to keep a vigilant watch on your query performance and resource consumption. It is a free a performance monitoring and tuning tool out-of-the-box in all versions of SQL Server 2016 and onwards. When I see my clients investing into very expensive third-party monitoring tools, I always wonder why they are not giving a chance to query store which is available out-of-the-box and does not cost anything in terms of money. I strongly suggest that you give query store a chance when you want to understand your query's past history. So far, whenever we talked about query plan, we always talked about actual execution plan. In the next module, we are going to compare that with estimated execution plan and also talk about live query statistics. Let's go to the next module, which we'll talk about query plan's past, present, and future.

Compare Estimated and Actual Query Plans and Related Metadata

Introduction

Hi, this is Pinal Dave, and I welcome all of you to a new module where we will discuss how to compare estimated and actual query plans and related metadata. This is a very interesting module because we are going to talk about query plans past, present, and future. First, we'll start talking about estimated execution plan. This plan is available before you run the query. Right after that, we will talk about actual execution plan. We have talked about actual execution plan in detail in this course so far, so we will just talk briefly about some of the properties of actual execution plan. Right next to that, we will also talk about live query statistics. This particular feature shows how exactly your query is running right now. This is a very useful feature, and I use it quite often in my SQL Server performance tuning consultation. Then we will talk about how to analyze execution plan. Please remember, in the previous module we have talked about how to identify poor performing query plan operators; now we will talk about as an execution plan what we can do with the help of SQL Server Management Studio. And finally, we will see real-world scenarios with the help of demonstrations. Well, that's what we are going to talk about in this module. Let's start discussing this module in the next clip.

What Is an Estimated Execution Plan?

Let's start this module from the quote of Charles Warner, "There is no such thing as absolute value in this world. You can only estimate what a thing is worth to you. " This is exactly applicable to execution plans. Execution plans and all the numbers and cost are relative to each other. If you look at any operators or any plan or any execution statistics, all of them are relative. They have no meaning if you just look at one number in absolute world. When you compare one execution plan with another execution plan, they start having meaning. And that is a very important concept for all of us performance tuning experts to understand. Now let's discuss about estimated execution plan. What is an estimated execution plan? The most probable plan the engine is likely to use when SQL Server generates an execution plan without running an actual query and display. Please remember, they keyword here is without running an actual query. So, SQL Server can give you estimated execution plan before running actual query and that plan most of the time is pretty accurate. Let's talk a little bit more about estimated execution plan. First of all, estimated execution plans are quicker because it does not have to wait for your query to run completely. If your query is 10

minutes long, to get actual execution plan, you have to wait for 10 minutes. But in the case of estimated execution plan, SQL Server will be able to give you that particular plan quite faster, or, as a matter of fact, in most cases, immediately. Remember, as actual query has not run, a similar execution plan does not contain any execution data. This is absolutely a no-brainer to us. Additionally, because it's an estimated execution plan, there is no resource consideration. For example, when you run your actual query, SQL Server has to see how much CPU memory and disk I/O is applicable for your query, whereas when you run estimated execution plan, SQL Server just has to base this decision on various past statistics and it does not have to consider any real-time situations. You can enable estimated execution plan with this shortcut, which is CTRL+L. Here are two commands which you can also execute to see estimated execution plan for your T-SQL in SQL Server Management Studio. Well, this is all I wanted to discuss about estimated execution plan. Before we go and see a demonstration of estimated execution plan, it is critical that we also see similar information about actual execution plan. Let's do that in the next clip.

What Is an Actual Execution Plan?

Now let's discuss about what is an actual execution plan? When SQL Server generates an execution plan running an actual query, considering available resources, it is an actual execution plan. Let's talk a little bit more about this execution plan. In earlier modules we have seen that actual execution plan is only visible after query is executed. And therefore, it contains it actual execution data. It also shows you how many rows were considered and what different operations it has to do to execute that execution plan. It also considers available resources. If you have more or less CPU, I/O, or memory, actual execution plan has to put them into consideration before running the query. The shortcut for actual execution plan is CTRL+M and here are two different ways we can enable actual execution plans. Well, this is a short discussion about actual execution plans. Now let's go to a very important section where we will see a demonstration about estimated execution plan and actual execution plan. Trust me, we have done a lot of work on actual execution plan, but now you will see when and where I use estimated execution plan to my advantage when I am debugging real-world performance tuning scenarios.

Demo: Estimated and Actual Execution Plan

Let's see a demonstration about estimated execution plan and actual execution plan. Though I prefer actual execution plan, more to do query plan performance tuning, there are good use cases where we have to depend on estimated execution plan. We will discover about them in this demonstration. Let's go to SQL Server Management Studio. Here we are in SQL Server Management Studio and we have two queries. The query on the top is very long query and it will take over 1 minute to execute, and second query is query with insert statement. We will discuss about both of them very soon. First, let's run this query by enabling actual execution plan. Click over here and now hit Run. Trust me, this query has started to display results immediately, but the query will complete after over a minute. I

don't want you to wait for 1 minute; hence, I am going to edit the part where you have to see this statistics screen. The query just completed and it took around 1 minute and 17 seconds. You can see the time over here. Now let's go to execution plan and click on Zoom to Fit. Our execution plan is in front of us. Let's find the most expensive operator in this execution plan. Here is a short operator, and it is taking 34% of the cost of this entire query. That is indeed very expensive, and this is the one which we need to tune or optimize it. There is yellow bang on this operator; that means there is a warning. Let's read what the warning is. The warning says the short operator is very expensive; hence, it has to short the data into TempDB. That's fair. Now we understood what we need to do in this query plan. We just have to find how we can remove this operator or create a better index. There are cases where we can rewrite this entire query and remove this short operator. However, rewriting query or creating indexes is not the scope of this module. Here we want to see what actual execution plan shows us, which is different from estimated execution plan. Now, this was an actual execution plan; hence, there are various details right below each of the operators. There are the number of the rows, as well as when you mouse over any of the queries, it also shows you some details related to actual execution. Though this is an actual execution plan, you will notice that SQL Server uses a lot of estimated numbers also in this query plan. So if anybody asks me, does actual execution plan display actual IO or CPU cost, I always say no. Actual execution plan is only going to show you estimated IO and CPU cost. So there you go, actual execution plan contains some of the estimated execution plan details. Now from actual execution plan, our takeaway is to fix the short operator. Now we will go and see what is the estimated execution plan of this query. Please note that my query completed in 1 minute and 17 seconds. That is not as bad, but there are cases in the real world where query takes hours or days to complete. In that case, it is impossible to wait for query to complete and show us actual execution plan. In those scenarios, it is better advice that we depend on estimated execution plan. Let me disable actual execution plan, and now we'll click on estimated execution plan. There are three different ways you can see estimated execution plan. First, is to use these SET commands. Second one is to go to Query and select on Display Estimated Execution Plan. And third one is by using shortcut or icon, which is in our menu bar. Let's select our query and click on this icon in Menu. As soon as we click it, it will display us estimated execution plan. And the estimated execution plan is pretty similar to actual execution plan. Though we do not see details about rows, it is very clear that sort operator is also very expensive in this query. Let's assume that this query is running for two hours, and we have to tune this query. It would be better that we check the estimated execution plan, take care of this short operator, and then we run this query with actual execution plan. So, we do not have to wait for the entire two hours. Estimated execution plan is quick and efficient where to see how your query is most likely to run. There are cases when actual execution plans and estimated execution plan is different, but most of the time, if you have updated your statistics regularly, you will find both of them pretty equal to each other. So, there you go. One of the biggest advantages of estimated execution plans is to get your execution plan immediately. Next one is about Insert. For example, look at this query. This is an insert query. If you have to find actual execution plan, you have to run this query and that would actually insert the data into table. In my example, I have used temporary table, but let's assume that this is a real table and if I run this query, it will insert real data into that table. This may not

be a good idea if we are debugging our query. In those scenarios, instead of looking at actual execution plan, estimated execution plan can be a blessing because actual query does not run and no data modification happens behind the scenes. Still, you can find your query cost and troublemaking operators. For example, when you look at this query, you know immediately the table insert is going to take 80% of entire query cost and that seems like most poorly performing query plan operators. Now without running query, we are able to figure that out. And this is really a good use case for estimated execution plan. You can also see estimated execution plan with the help of this set command. Let's execute SET SHOWPLAN_ALL and now run our query with insert. When I run this query, estimated execution plan immediately displayed in the grid with various details related to different operators. If you want estimated execution plan in XML format, you can easily do that as well. First, turn off SHOWPLAN_ALL and next enable SET SHOWPLAN_XML. Now when you run any of the queries, it will show you execution plan into XML. You can copy this XML and send via email or share via any tool with your colleagues. You can click on any of these XML and see your execution plan. Let's check that out with this long-running query. For example, when I run this long-running query, it displays us XML plan over here. Now when I click here, it will bring up same execution plan which we have seen earlier. Well, by running SET SHOWPLAN_XML OFF, you can turn off estimated execution plan. With this demonstration, I'm very confident that you learn two of the real-world scenarios where estimated execution plan can be helpful over actual execution plan. So now we can say estimated execution plan is the past of a query and actual execution plan is the future of the query. In the next clip, we will discuss about live query statistics and that is about current status or present status of the query.

What Is Live Query Statistics?

In the previous clip, we discuss about estimated execution plan and actual execution plan with a demonstration. Now we will discuss about what is live query statistics. This is a very interesting feature introduced in the latest version of SQL Server. The live query statistics displays the real-time insights into query execution process, as well as control flows of the operator. If I were to go back a few years, I always struggle to tune live queries. I always have estimated execution plan, which is generated before query and actual execution plan, which is generated after the query, but I never had a tool which helps me to see what's exactly going on in the query which is running currently. Now with the live query statistics, that particular problem is resolved and it is very efficient and helpful. I use this particular feature when I have to figure out what is running on the server which is slowing down my application. Let's talk a little bit more about live query statistics. Live query statistics is visible while query is executed. That's why it contains actual execution data. However, as the query is not completed, it will have partial actual execution data. It also considers available resources and you can see live query statistics from Activity Monitor of SQL Server. If you ask me personally, live query statistics is one of the most under-utilized feature of SQL Server. Let's go to the next clip where we will see with the help of demonstration how live query statistics works.

Demo: Live Query Statistics

Let's see in this demonstration how live query statistics works. Let's go to SQL Server Management Studio. Here we are in SQL Server Management Studio, and I will take help of the long-running query, which I had run earlier in the previous clip. Before I run this query, let's go and enable live query statistics. You can do that by clicking on this particular icon or going to Query and selecting the option of Include Live Query Statistics. Unfortunately, there is no keyboard shortcut for the feature. Once you select this option, click on Execute for your query. Once you click on Execute, it will start showing you live query statistics. Here it is. Let me select zoom to fit, and now entire execution plan is in front of us. This execution plan is actually real-time data. We can learn a lot about our queries by just looking at this data. For example, if you were thinking this cluster index was very expensive, that's not true because execution of this operator was just completed in 0.4 s. The sorting which is going on here, is actually taking a lot more time, and that's what is displayed over here. Currently, the sorting has reached to 70% and it is increasing as we talk. Similarly, data which is passed via various operators, is also visible in the screen. This is very interesting data if you are into query tuning. Now we will explore how we can learn about live query statistics with the help of activity monitor. Let's select this activity monitor and now it will bring up this screen. Further expand Active Expensive Queries. Now go and run our long-running queries. As long-running query is running, you can go back to our Activity Monitor and wait until it shows up under Active Expensive Queries. Now you can right-click over here and also see live execution plan. This is a very handy feature. A lot of people have no idea that you can go and see live query execution data from activity monitor. Activity monitor is a very tool for SQL Server for performance tuning, but just like live query statistics, it is also heavily under-utilized. I hope this quick demonstration enables you with additional weapon to tune your poorly performing query plan operators. Analyzing query is a very interesting subject and there are many tools available natively in SQL Server Management Studio. We will discuss about three of the important tools in the next clip.

Demo: Analyze Execution Plan

A lot of people depend on third-party tool to analyze execution plan. However, SQL Server Management Studio is very efficient and it has three cool features to help you analyze query execution plan. The first feature is show plan analysis, the second one is compare showplan, and third one is find node. Let's see how each of these works with the help of SQL Server Management Studio. Here I am in SQL Server Management Studio, and first I will run a sample query. When I run this query, I will leave actual execution plan enabled. Now click on Execute. When the query completes, it will bring up Execution plan tab over here next to Messages. Now click there and right over here, right-click on any area and select Analyze Actual Execution Plan. Analyze Actual Execution Plan will bring up this particular screen where we will see option to select our queries. As we have run only one query, this is the query which we can analyze. Select scenario and it will bring up very interesting information about our query. We can

clearly see our query is inefficient. We already know there is a missing index warning, but that we knew from query plan. However, when we come to Showplan Analysis over here, we can clearly see a difference between what SQL Server had estimated for our operator and what was the actual number. This clearly indicates that our data, or rather, statistics data for our database is very old and we need to go and update our statistics. Once we update our statistics, for every single table in our database, we will start having better estimation and eventually that will bring us better actual execution plan. Now you can clearly see that SQL Server itself calls out what's exactly wrong with your query plan, and we do not have to depend on any third party tool for the same. Next, we will talk about compare showplan. For the same, we need to use actual execution plan, which we have generated just now. Let's close this section about showplan analysis and right-click over here and save execution plan as. I am saving this execution plan as firstplan. Click on Save and it has saved this execution plan. Next, go to any other query and run the same query. Please note this query is a different query and has a different where condition. Click on Execute and now it will bring up execution plan over here. The execution plan of this query is very different. If you want to go back and compare execution plan of this query with previous query, you can easily do that. Right-click over here and click on Compare Showplans. It will take our current plan and any plan which we select from our hard drive. Here I will select our first plan, and there you go, it will bring up comparison between two of the plans. We also have Showplan Analysis once again here where it gives us option to highlight similar operators. Both the queries are listed over here as a top plan and bottom plan. Let's close this one and on the right side we can see difference between various operators. For example, we have Clustered Index Seek on InvoiceLines, and the cost is 78%. However, the same Clustered Index Scan is now 28% in the second case, and that is visible in this box over here. There is various information related to each of the operators displayed on the right side. When you select any operator, it will bring associated information on the right side. For example, if I select Invoices table, SQL Server will bring Invoices table on the left side and Invoices table on the right side. One the left side, there is a NonClustered Index Scan, and on the right side, we have a Clustered Index Scan, and here are all the differences between both of these operators. This can be very, very helpful when you are doing benchmarking, as well as trying to get last bit of performance from your query. I use this thing very extensively once I take care of low-hanging fruit. There is one more feature we still have to discuss, and that's find node. Let's click on Close, and now let's go back to our query. Here is the execution plan of our query, and right-click here and select Find Node. It will bring up a small drop-down on the top of it which says Actual CPU in milliseconds. Now here I can find a lot of different properties about our execution plan and we can go back and find associated information about them from our query. For example, let's go and select CPU time and say CPU time contains 100. That means anywhere where CPU time is 100 will be highlighted. In our case, there are none. So, let's select some other data. Maybe there is a key lookup, and we should say select lookup to true, and when we click Go, it will bring up any operator where we have lookup is equal to true. For example, this is the only operator in our execution plan where lookup is equal to 1. Well, now we have seen three important native SQL Server Management Studio features which can help you to analyze your query plans, you do not have to invest into

third-party tool to just do this simple task because you can get them done right away from this easy interface. Well, with this, we come to the end of our module, and let's do a summary of this module in the next clip.

Summary

In this module, we discuss about various available features of SQL Server Management Studio which enables us to look into query plan execution. We can see query plan's past with the help of estimated execution plan and future with the help of actual execution plan. Current state of query plan is available with the live query statistics. Additionally, we have used quite a lot of different tools to understand how query plan works and what are the features available in the SQL Server Management Studio. I want to summarize this module with a single line. While query performance troubleshooting, it is critical to use different query plan tools to find the troublemaking operators. You should be brave enough to do various experiments to find out which particular operator is slow and how you can optimize that. Well, so far what we have discussed is about in-premises SQL Server. They definitely are applicable to Azure VMs and AWS VMs. Now we will go to our next module where we will talk about configuring Azure SQL database performance insight. That is a native tool out of Azure to help you understand performance of your SQL Server database.

Configure Azure SQL Database Performance Insight

Introduction

Hi, this is Pinal Dave, and I welcome all of you to this module how to configure Azure SQL database performance insight. So far, in the previous module, we have talked about how we can do query plan analysis on in-premises SQL Server. All the information we learned can be also applied on Azure VM or any stand-alone installation of SQL Server. However, when we talk about Azure SQL database, things a little bit change, and thankfully Microsoft has released Azure SQL database performance insight. Let's talk about agenda of this module. First, we'll talk about what is query performance insight, and right after that we will see a few of the use cases. Right following that, we will discuss about real-world scenarios and demonstrations. This module is a little bit smaller than previous module, as every information you learned in the previous module can be applied to this module as it is, but this is added information if you are using Azure SQL database. Well, let's go to the next clip and start discussing query performance insight.

Query Performance Insight

Johan Cruyff said, "Speed is often confused with insight. When I start running earlier than the others, I appear faster. " This is how query performance insight works with Azure. If you know what's exactly going on under your system, you always make informed decisions, which seems to be efficient and faster. Let's understand what is query performance insight? Query performance insight provides deeper understanding of the database resource consumption and details on inefficient queries. This is one of the features which I truly appreciate in Azure SQL databases. Query performance insight is one of the very valuable tools if you are using Azure. Let's learn a little bit more about this feature. First of all, this feature provides us deeper insights into our database resource consumption. That is DTU. DTU stands for database throughput unit. It is a blended measure of CPU, memory, database reads and writes. DTU is also directly related to cost of your Azure database. If you use a lot of DTUs, you have might to pay more money to Azure based on your subscription label. This is why it is very critical to know how much DTU your queries are consuming. Next feature is about top queries by CPU, duration, and execution. Query performance insight lists all of these top queries and gives us opportunity to tune them. Remember, everything we discussed in the previous modules about identifying poorly performing query plan operators and tuning them absolutely applies

over here as well. You can take any query, analyze their query plan, and find a way to optimize that particular operator. If you ask me about homework of this course, I would say go and find top 10 queries by CPU, duration, and execution, and start tuning them. Once you are done with the task, you may get amazing performance for your database. Another very useful feature of query performance insight is history of resource utilization. It gives us the ability to drill down into details of query and view query text along with its resource utilization. This is very helpful if any of your query suddenly starts taking more resources than what it is has consumed previously. We can identify those queries and operators and see what's going on inside our system. Performance recommendations, this is one of my favorite features of query performance insight. SQL Azure gives us performance recommendations at query level. It informs us what we can do with our query to get maximum performance out of it. It may be an index recommendation or suggestion to rewrite query. Use performance recommendations as your starting point to tune your SQL Azure queries. I find this feature very helpful when I have no idea where to start tuning my SQL queries. And finally, it is very important to enable query store on SQL Azure to take advantage of query performance insight. Query performance insight requires that query store is active on your database. Remember, on every single Azure SQL database, query store is enabled by default automatically behind the scenes. If query store is not running, the Azure Portal will prompt you to enable it. If it is disabled, you cannot use query performance insight. Well, that's a lot of information about query performance insight. Now let's go to the next clip where we will see a demonstration about query performance insight on Azure Portal.

Demo: Query Performance Insight

Let's see a demonstration about query performance insight. For a change, this time instead of going to SQL Server Management Studio, we will go to Azure Portal in browser. Here I'm logged into my Azure Portal and this is the home screen. First, let's go to the left side and select SQL databases. SQL databases are here on the top in my screen. If you do not have SQL databases over here, you can search over here. Next, click on SQL databases and now select the database which you want to inspect. I'm going to select WideWorldImporters. Once I select the database, it will open secondary menu on the right side of our main menu. I'm going to minimize our main menu so we get a little bit more real estate on our screen. Next, scroll down in our secondary menu and select the option of Query Performance Insight. Once I select Query Performance Insight, it will bring up the screen with three tabs. The first tab is about resource consuming query, the second tab is about long running queries, and the third tab is about customization. Let's talk about first resource consuming query. Currently, the option of CPU is selected. That means this screen will show you top 5 queries by CPU. Scroll on the right side, and here are all the five queries which has taken maximum CPU in our system. The very first query has ran around 854 times and has taken around 0.9 % of the CPU. The total time all these queries have taken around 2 minutes and 26 seconds. Isn't this surprising that query has only taken 2 minutes and 26 seconds, even though it has used 0.09 % of CPU? This is definitely worth looking into query because this query may look very, very innocent, but when it runs in multiple numbers on our

system, it takes a lot of CPU. Here is the query which is taking a lot of CPU. You can select the query text by going to this option of query text. Copy the query text and close this window. Now we will go back to our SQL Server Management Studio and paste our query text. Please note that my SQL Server Management Studio is currently connected to my Azure instance. Let's run this query by enabling actual execution plan and see what it has to say. When I execute this query, the execution plan immediately showed up because query runs pretty quickly. Upon looking at execution plan, it's very clear that it is missing an index, which can improve this query's performance by 95%. This is the reason why this query is so expensive even though it runs super fast. Once we will create this index, I'm very sure that our query will not come up in the top 5 CPU consuming queries. Now let's go back to our Azure Portal. Here click close on this query. Now we can inspect any other queries which are listed here. Instead of looking at those queries, let's go back on the top and select on other query metrics, which is data I/O. Once we select data I/O, our graph on the right side changes, and when we scroll down all the way to the bottom, we can once again see different queries. The query on the top right now has taken 0.1 % of CPU. That is way lesser CPU than this query, and this particular query has run only once. Even though this query has run only once, it has read a lot of data because the query had ran for almost 5 hours and 19 minutes. Definitely this query seems to be very, very expensive. I wonder what developer would have written in this query that it has not taken much of the CPU, but read lots of data. Let's select this row and see what the query text looks like. Here we go, our query text is in front of us and there are three cross joins. Cross join is one of the most expensive operations, and I don't see any reason for this kind of cross join in our application. I am sure a developer might have just run a test query and would have forgotten to just close SQL Server Management Studio. There are also no where condition, and query would have read a lot of data from our database. If you want to see at what time this query completed, you can scroll down all the way to the bottom, and it will show you the last 24 hour history. Over here it is very clear that query finished at this particular time after running for 5 hours and 19 minutes. That's why when the query completed, it recorded its execution time along with how much CPU it has consumed and execution count. This is very helpful because now I can go back to 5 hours and 19 minutes before this time and check with developer what he or she was doing with the server. Let's scroll down and now we have queries once again ordered by our log IO. Here is another query which has just ran for 5 s and 870 ms. Even though it has taken around 0.1 % of our log, this query seems to have a nature of updating something in my database. This query also has executed around 710 times. Let's go and check what this query looks like. Upon selecting the row, it brings up an insert statement to us. And here is an insert statement. Insert statement will increase the log IO and my assumption was correct. Let's scroll down to see at what different time our queries were executed. When we scroll down, we can clearly see that pretty much all 710 queries were ran between 7:00 AM and 8:00 AM. They just created a lot of IO. Now, click Close over here, scroll up, and click on long running queries. This is the second tab in our query performance insight. Upon scrolling down, now this screen will show us all the queries which have run for a longer period of time by duration. Please note that in the first step, we've discussed about CPU, data IO, and log IO, and now we are focusing on duration. We have already inspected the first query and also inspected the last query in the list. However, we have not looked at the other three queries, so let's

select third row and understand what this query does. This query is again a cross join and it has executed around five times, as we have seen in the previous screen. The history says that it has executed 5 times between 6:00 AM and 7:00 AM. Click Close over here and we are back into our main screen. As you can see, with the help of query performance insight, we can identify our resource consuming queries and long-running queries very, very easily. We can also go back into history of our query and identify what exactly was going on at any particular time. When we click on Custom section, we can further customize our screen based on what exactly we want to see. We can change metric type to any of the metric types listed in the drop-down and also change the time duration. This particular drop-down gives you control on selecting how many queries you want to see in your main screen. And the final drop-down talks about what particular kind of aggregation you prefer to see. I always like to see the sum, as that gives me a more comprehensive view than max on average. Well, this is query insight for you. However, before we stop talking about it, let me point you to one very important aspect of intelligent performance of Azure Portal. Azure also provides you automatic tuning opportunity. When you click on this tab of Automatic tuning, it gives you three different options to automatically tune your system. You can configure each of these options independently and let SQL Azure decide what they want to do to tune your queries. This is a very powerful feature and I strongly suggest that you explore various offerings of SQL Azure Portal. Now we have seen a demonstration of query performance insight, let's go and talk about what else you could do with Azure Portal.

Going Beyond Query Plans on Azure

So, we just talked about query performance insight. However, SQL Azure provides many different tools which helps SQL Server performance tuning expert to tune any query on SQL Server. For example, in the previous demonstration, we have looked at automatic tuning. You can configure automatic tuning to solve many of the common database performance problems. I always advise all my clients to leave automatic tuning on in the beginning and decide if they want to turn off or not later on after doing thorough testing. The second one is intelligent insights. This is also available on only Azure database, and it lets you know what is happening with your SQL database in terms of performance. It uses built-in intelligence to continuously monitor your database usage with the help of AI and informs you about any anomaly or disruptive events. Trust me, if you are using Azure, this is one of the things you may want to consider using it. The reason we are not covering this in this module is because it does not help us to analyze our query plans, which is the primary goal of this course. Here is one more feature which is only available on Azure, Azure SQL Analytics. If you have lots of SQL database, elastic pools, and managed instance, this is a must-have tool for you. This goes beyond query plan and looks at your SQL database at instance level. I strongly suggest that you consider Azure SQL Analytics if you have lots of database to manage and have a little time on your hands. Azure is continuously evolving. Along with query performance insight, if you use all these three tools which are listed on the screen, I'm very confident that you will get the best out of your SQL Azure database. Well, with this, we come to the end of our module. Let's jump to summary slide in the next clip.

Summary

In the previous clip, we have discussed about what else we can do on SQL Azure along with query performance insight. Now is the time to summarize this entire module. Remember, everything we discussed in this module applies to Azure platform; however, all the other modules before this module applies to SQL Azure as well. Performance tuning with the help of analyzing query plans is one of the very critical skills everybody should have. Microsoft has done a fantastic job by taking this entire experience to the next level by introducing query performance insight in SQL Azure platform. If I were to summarize in one line, this is what I would say. Query performance insight helps to understand the impact of query workload and its impact on resource consumption. On Azure, you want to take care of your resources. Well, with this, we end this module and now we will jump to our very final module where we will talk about what you can do once you are an expert at analyzing query plans.

Summary

Summary

Hi, this is Pinal Dave, and I welcome all of you to the very last module of this course, Analyzing Query Plans. Let's read a very popular quote from Malcolm X, "Every defeat, every heartbreak, every loss, contains its own seed, its own lesson on how to improve your performance the next time. " If you remember this statement, I promise you that you will be an excellent SQL Server performance tuning expert. Every time you look at any query plan, you know there is a learning opportunity in it. Let me simplify this statement with another quote. When there is a SQL Server performance issue, our primary goal is to find the root cause of the slowness. Well, there is no author listed here because this is my statement. Once we know the root cause, solving the problem is very easy. There are a set of the problems and set of the solutions. It is very important that we know that what's the right problem and what is the real problem of our performance issue. If you make a mistake in finding root cause, you may end up in a situation where you solve the problem which you don't have it. Analyzing SQL Server query plans is one of the most important tasks for any DBA or developer. Thankfully, Microsoft have made it very easy for us by providing us many tools. Here are all the major components of analyzing SQL Server query plans in a single glance in front of you. Extended events, query trace or profiler, query store, query plans, expensive operators, and query performance insight. Once you apply all of this together on your SQL Server query plans, you start understanding how they work and what you can do to get maximum out of your query plans. Now it's time for me to say goodbye to you. Let's stay connected with each other. If you have any SQL Server performance trouble, you can always post your question at Pluralsight discussion forum. You can also reach me at pinal@ sqlauthority.com. I do my best to answer every single email in 24 hours. Please ask me any questions and I would be happy to answer you. With this, we end this course. Thank you very much for joining this journey to analyzing query plans.