Course Overview

Course Overview

Hi everyone. My name is Viktor Suha, and welcome to my course, Managing SQL Server Database Performance. I am currently a database developer and DBA at GSGroup. Have you experienced performance problems with your SQL Server database solution and had a challenge to identify the root causes and resolve them? In this course, we are going to explore what external factors can adversely impact the database performance, even if your workload is seemingly optimized. Also, we are going to cover the troubleshooting and optimization methods to either prevent or resolve these problems. Some of the major topics that we will cover include the importance of server maintenance, including patching SQL Server; optimizing SQL Server memory settings and tempdb configuration; estimating Azure SQL Database service tiers and sizes; troubleshooting common performance problems with wait statistics. By the end of this course, you'll know how to prevent common SQL Server performance problems from happening in the first place and how to troubleshoot performance problems more efficiently. Before beginning the course, you should be familiar with using SQL Server Management Studio to be able to run queries against any version of SQL Server. From here, you should feel comfortable further diving into performance troubleshooting and optimization with courses on SQL Server Wait Statistics, Automatic Tuning in Azure SQL Database, and Using the SQL Server Query Store. I hope you will join me on this journey to learn about performance troubleshooting and optimization with the Managing SQL Server Database Performance course, at Pluralsight.

Aiming for Performance and Scalability

Version Check

Introduction

Hello, I'm Viktor Suha, and welcome to my course, Managing SQL Server Database Performance. I'm a database developer and SQL Server DBA, having many years of experience with the Microsoft database platform. Over the years, I worked with both developers and infrastructure specialists to troubleshoot and optimize production SQL Server environments. I got to realize that the wide gap between developers and IT infrastructure people has by now grown even wider and poses a serious challenge in addressing performance-related problems on a daily basis. My intention with this course is to help you narrow this gap. So you have returned your first queries in T-SQL, you have returned stored procedures, functions, views, and find that sometimes your queries, or how we often refer to them, the workload tends to run slower than you expect, even in your own development environment. You might work with frameworks like Entity Framework, where you don't even know what the workload will look like under the covers. Everything is managed by client-side code; still you have performance challenges. You might also have been working with SQL Server for a long time now, but how many times has it happened to you that your customers or users started to complain about your slow application, its sluggish performance, timeout exceptions, and everyone was pointing to the database? They claim that the application used to work okay, but suddenly it has become unresponsive, queries have become slow, even though reported and that nothing got changed. It happens too often with production systems, actually. When speaking with database performance, we tend to think about slow running and badly written queries in particular, and all efforts are focused on capturing these and resolving coding problems. While this is absolutely valid since query design and quality are crucial, there are still many other aspects in a database environment that can impact workload performance. There might be a problem with the host environment or infrastructure, hardware and network issues for example, might be a problem with server configuration like non-optimal SQL Server memory settings or wrongly configured tempdb. Might be a problem with the database itself, unexpected database settings, bad schema design, missing indexes or bad indexes, and the database can even be corrupted. There might be a problem with the data, data size has grown tenfold, data distribution has changed significantly, or unexpected data values found their way into the database. Also operations and maintenance may have missed crucial alerts in monitoring, or there is no monitoring at all. The server has not been patched for a long time now, or important database maintenance jobs like index maintenance, are not running. Can we isolate the

problem to one or more layers somehow? It is often a combination of all these in production systems, actually. How do I know where the problem is in the first place?

A Holistic and Layered Approach

In this course, I teach you a holistic approach to analyze, troubleshoot, and resolve performance-related problems within the Microsoft SQL Server Database environment. I focus on the most common types of problems and introduce troubleshooting patterns. Besides addressing all the existing problems in the database, I also teach you methods like a server configuration health check that potentially prevents major problems from occurring in the first place. The approach I teach you within this course can also be considered a layered approach. What does this mean? Remember the various factors that might impact our workload performance, the environment, server configuration, database settings, and our workload itself can have problems too. Imagine these as layers on top of each other, no matter what environment we are in, on premise or in Azure. My aim is to identify which layer or layers the problem comes from and target that layer specifically. Also, I can go top down or drill down from a high level in the environment to find details in our queries. I raise the question how to get started. Imagine the entire process as a tree. This is a tree in wintertime, it seems, no leaves, which has many branches. Traversing all branches would probably be quite widespread and overwhelming without guidance. I want to identify troubleshooting entry points, a branch which I can then follow in more detail. Each branch might require multiple disciplines and skills. If I find the root of the problem in the host environment, for example, with the underlying storage itself, I will need to communicate with other people who are experts in that field, own that technology, and can help me. One branch leads me to workload traces and career plan analysis, for example. The other verifies database settings, server configuration settings and behavior, maintenance tests, changes in data, or environment-specific configurations like sizing of a VM or tier in Azure. So I better try to narrow down the problem to particular branches to save everyone's time and, at the end of the day, keep the costs at bay. If we don't have an approach like this, essentially we are just shooting in the dark and collecting data from everywhere with whatever tools we can without a clear purpose.

Definitions and Methods

But what does performance and scalability mean in the first place in regards to a database environment? Let's first introduce yet another term, SLA. In production environments, we often work with SLAs. The environment itself has a predefined SLA, or at least it should have one. It more or less defines availability, for example, how long down times a system can have in a certain time period. But for us here, a SQL Server workload can have SLAs too, once the allowed maximum duration of the workload to provide the expected results or how responsive a particular database application functionality is, as expected by its users. Performance itself is often perceived as how fast the workload or the application is. However, unfortunately, it is often just a perception and not an exact measurement, as it should

be. Scalability is a highly important aspect to database workloads in general. It more or less means in practical terms, that I start working with a smaller set of data or number of concurrent users, and my database solution is performing well in predefined SLAs. Over time, when data grows tenfold and the number of users increases with hundreds more, my solution still performs well under the predefined SLAs. In case it's needed, I can also adjust the system configuration relatively easily to accommodate the higher usage patterns without major downtimes. I don't need to buy new hardware or buy new storage or migrate into a totally different platform. This applies to the database application and the workload too. It really does matter what the database schema design looks like and how efficiently I've written my code so that it scales. It's important to understand and analyze performance and scalability with diagnostic data, not just to rely on perception. It also serves a very useful purpose. By having baselines, we know how our database environment performs and behaves, even when there is no perceived problem. When the actual problems start to occur, as reported by users or the monitoring system, we are then able to better identify the changed patterns and compare the erroneous behavior to our baselines.

Aiming for Scalability and Performance

If we aim the endpoint quality of our SQL Server database solution having performance and scalability in mind, I could list six areas to focus on. Understand the business requirements in depth. What is expected from our solution exec at the business level? Solution planning and design with SQL Server-specific coding and application development patterns, best practices and technology choices. Usage patterns to understand how our database solution would be used by the users at the data level, how the load will change over time. Infrastructure planning to plan and size the entire database environment infrastructure properly so that it performs and scales well. SQL Server configuration, of course, to configure our SQL Server properly according to the particular environment and workload patterns. Operations and Maintenance. This makes sure that we can prevent major problems from occurring, as well as we can react to problems properly in time. Due to our limited scope, I don't cover the first two areas within this course. Now let's see what usage patterns mean. Data growth trends. How data is expected to grow over a period of time. Can my workload still perform when data size has grown? Have I tested with relevant data sizes? Data table sizes. Which are the largest tables in the database? Are the sizes justified, or can I archive unused or cold data? Data distributions. Do I have unusual data distribution patterns? How does it affect my indexing scheme? Have I tested with relevant data? Do I need to test with production data? Concurrent users and peak times. Do I know how many users will use my solution? Can the number of concurrent users increase significantly? Are there peak usage times? Have I done relevant load tests? Infrastructure planning. Within infrastructure planning, the primary aim is to choose the most appropriate technologies, order, with performance and scalability in mind, and build the environment in a way that it performs and scales according to the business needs. Many times it means sizing service properly or choosing the proper tiers in Azure. SQL Server configuration. I cover SQL Server configuration and details in later modules. It's crucial to know the particular SQL Server platform you are working with, both as a developer and an

administrator. By doing regular health checks, you can prevent major incidents, apply best practices, and recommendations. Also, you can document and track what's happening in your environment. SQL Server configuration settings are not carved in stone for good. If conditions change, the settings might need to be adjusted. Operations and maintenance. This is an area that is often simply missed due to lack of skills or "cost factors. " It is crucial to have someone in charge with the owner of the environment to have control and the responsibility. Normally, that person is a dedicated database administrator, or DBA, having technology-specific training and skills. I already highlighted the importance of ongoing monitoring and maintenance. Ideally, there is a SQL Server-aware monitoring solution in place too. There are a couple of such solutions available on the market, but you still require the skills to understand the diagnostic data to be able to diagnose the problems in depth. Ownership of the database environment inherently should mean the ongoing maintenance of the environment, which ranges from patching the server to running index maintenance and database integrity checks, among others.

Scope and Assets

The methods and approaches I show you within this course generally apply to all supported versions of SQL Server currently; that is SQL Server 2012 and onwards, up until the very latest release of SQL 2017. My demos within this course have been written and run against SQL 2017 specifically. Some of the tools and features highlighted might apply to a particular newer version of SQL Server only. I will make note of those each and every time I cover such feature. The more recent version of SQL Server we are working with, the better the built-in instrumentation is to help us with performance, troubleshooting, and optimization. For example, when I talk about the Query Store, that applies only to SQL Server 2016 and above. The main concepts discussed, like wait statistics, however, are rooted back to older versions of SQL Server R2, way back to SQL 2005 even. So in case you happen to deal with such old systems, which are officially out of Microsoft product support by now, thus grayed out in this slide, you can still apply some of the methodology learned here. Besides an on-premise full-fledged SQL Server at the core of our course demos and discussions, I also teach you concepts within the Azure SQL Database environment in Microsoft Azure, dedicating a separate module just for that. My demos and code samples utilize the WideWorldImporters sample database that is publicly available for download at the Bitly link above. I use the SQL Server Management Studio, or often referred to as SSMS client environment, for server management and coding demos. SQL Management Studio is the ultimate client tool for managing SQL Server environments. Since our course covers multiple aspects of performance troubleshooting and server configuration, this tool has everything we need. SQL Management Studio is now a separate tool and has its own release and update schedule, so I would encourage you to always keep it up to date and check up on the very latest release periodically at the Bitly link above. As a matter of fact, managing database performance, performance troubleshooting, and optimization is more like an art. SQL Server is one complex piece of software. It's like a highly complex instrument. You have a complete technical arsenal at your hands, top-notch instrumentation, and diagnostics built into SQL Server, yet the challenge is, rather to understand the problem,

understand the context and identify entry points in your process to be more efficient, essentially to save time and money.

Overview

To close our introduction, let's have a quick review of what you can expect from this course in the next modules. I talk about a few SQL Server concepts that are crucial to know in order to dive into performance optimization. I talk about versions, but also about the threading model, which leads us to the key concept of waits and wait statistics that essentially serves our intention to find optimization entry points. I go through a SQL Server health check, what to check and configure in advance in a SQL Server deployment to prevent major problems from occurring later on. I dive into tempdb and database file configuration options, somewhat to continue our server health check process. I show you two offerings in Microsoft Azure, SQL Server on VMs and Azure SQL database, the latter being a database-as-a- service offering, and talk about the key differences, but also the similarities, plus discuss what our options are for performance optimization in those environments. I show you tools and methods that you can use for baselining and troubleshooting your SQL Server database environment. We will see what it's like in action that we have learned so far. Database performance management can be highly complex and overwhelming sometimes, and we can only cover a fraction of it within this course. But we need to start somewhere, so let's get started.

Understanding Key SQL Server Concepts

Major Versions and Compatibility Levels

Hello, and welcome back to my course, Managing SQL Server Database Performance. In this module, I talk about a few SQL Server concepts that are important to understand for server and database environment configurations, also to get started with performance troubleshooting and optimization. Some of these concepts discussed here are simplified for getting a high-level overview first, but I will also revisit these concepts in more detail in later modules. I encourage you to dive deeper in any of the topics shown here, and it also takes a lot of practice and experimentation in your own environment to see how these things really work. No two environments and workloads are the exact very same, so as it's often said in the SQL Server community, welcome to the world of it depends. In this section, I talk about on-premise SQL Server versions, editions, and the importance of patching the server. When discussing SQL Server environments, and also to start with any kind of troubleshooting and optimization, the very first thing that we want to know is the exact version of the SQL Server platform we are working with. It's that important that it determines many things. No matter how trivial it sounds, but knowing the exact version number might also cut the entire troubleshooting process short and save us an awful lot of time and effort. The first component of the version number is the major version of SQL Server. As of today, the list of supported major versions is SQL 2012 up until the latest release of SQL 2017. SQL 2019 is coming soon, which is already in CDB phase. As you see in the timeline, major versions are also indicated with an increasing number like 11, 12, 13, or 14, and also sometimes referred to as v11 and so on, corresponding to the major version number. For example, SQL 2017 is 14.0 and v14. As we go from older to newer releases, the supportability and servicing by Microsoft changing and improves. Regardless of this, you might still see old and officially unsupported versions out there that are grayed out in this timeline. What does it mean that a SQL Server major version is supported or unsupported anyway? It is supported or unsupported by Microsoft officially as part of their product lifecycle. When a new major version is being released, it begins its product life that spans 10 years from then on. After that, it reaches its end of life, or the product expires. During this 10-year lifespan, the release is getting support and has servicing from Microsoft, batches and fixes arrive, and customer support will help you in case of an incident. But this lifespan is divided in two. The first 5 years is the mainstream; the second is the extended support phase. The servicing scheme between the mainstream and the extended phase is different. While you get functional fixes in the mainstream phase, the extended phase provides you with security or other critical fixes only. In general, the highest servicing efforts go into releases that are in mainstream support phase. You can check the exact lifecycle details for any major versions or any other Microsoft products, for that matter, in the search product lifecycle you are shown above. Let's see what implications of the major version has in our database

environment. Each major version introduces a new corresponding database compatibility level, or comp level, in short. The comp level provides new behaviors, improvements that come with that major version in particular. For example, SQL 2017 introduced comp level 140. SQL 2016 introduced 130, and so on. In the meantime, for backwards compatibility, which is especially useful when migrating databases from older to newer versions, each major release supports a list of older comp levels too. If your database runs with an old comp level and gets migrated to a newer server when that level is not supported anymore, like SQL 2005 comp level 90 is migrated to SQL 2014 or above, the comp level will automatically change to the lowest available level in that version. Again, why this is highly important, because the comp level determines database and workload behavior. Let me show you a few examples of key performance-related and behavioral changes between comp levels 110, that is SQL 2012, and 120, that is SQL 2014. SQL 2014 introduced a new version of the cardinality estimation, or CE, an important part of the Query Optimizer. This change can directly mean different execution plans for the very same workload for the better or worse. Also, if an older and migrated workload running on SQL 2014 with a comp level changed, the 120 produces unexpected performance problems, and it can happen with any later releases too, you can take it as a red flag all ready to help you narrow down the source of the problems. Similarly, a few key changes between comp level 120, that is SQL 2014, and comp level 130, that is SQL 2016, are shown above. Again, changes in the CE and also a behavioral change, with trace flag 2371. This is impacting statistics updates. Back in SQL 2014, you needed to use the trace flag. In SQL 2016 with comp level 130, this behavior is built in.

Patch Levels and Servicing

Now let's talk about servicing of a major version with an example, SQL 2016, in this case. When the product is released, it's called RTM, or release to manufacture. It comes with an RTM patch level. Then over time, corresponding to the servicing model, security fixes, functional fixes, within cumulative update packages, or CUs, are being released, first still in the RTM branch and the build number or batch level increases. Then comes the first and any subsequent service packs, or SPs, with their branches of fixes. As of speaking for SQL 2016, CU6 on top of SP2 is the latest patch level. This SP2 branch is a supported branch. Anything lower than that raises supportability questions and makes the product vulnerable from a security perspective, potentially, and of course many other fixes are missing too. How to check the exact version number of SQL Server you are using? There are many options. One of these is using @@VERSION function, as in the code example, after connecting to the server. It returns a concatenated string of data. Make note that in different major versions or patch levels, you might get varying levels of details for versions with this method. Similarly, you can use the SERVERPROPERTY function with separate parameters, too, to get the desired version information in a tabular format. For checking up on version, the release notes, servicing updates, your primary source of information should be official Microsoft Knowledge Base, or KB, articles, besides the blogs from the product groups. When searching for any known issues or updates for SQL Server, just bring up your favorite search engine and type in the search term, along with site: support.microsoft .com

to directly within KBs. If you wonder what versions have been released so far and what build numbers, patch levels, have been available, I would encourage you to check up on the mentioned KB321185 article at the URL above. While my servicing example showed a typical service pack-based model for SQL 2016 and below, coming with SQL 2017, a major update was introduced by Microsoft. No SPs will be released for SQL 2017 and above as part of the modern servicing model. SPs will still be released for earlier versions. It is now fully recommended to patch your service proactively with CUs in an ongoing manner, regardless of major version. Why does the major version of SQL Server matter for us here when speaking of performance, troubleshooting, and optimization? The major version comes with a set of features, obviously, that might cause the problems, or we can use for our advantage and problem resolution. It determines workload behavior, for example, via DB comp levels. It really does matter what tools and methods we can use to troubleshoot the problems, what our configuration, or even workaround options are. As we have seen, it also determines how the product is being supported and what our patching options are. With some experience, we will know what are the most effective troubleshooting and optimization options, given the particular version number. Why does the patch level, or build number, matter? Fixes, whether in service packs or cumulative update packages or standalone updates, will provide security fixes, which is a very important factor, bug fixes, performance-related fixes and improvements, even new functional improvements and behavioral changes. Let's see an example what CU14 for SQL 2017 bring for KB article 4484710. If you're in production, you really do want to consider these updates in an ongoing manner.

Editions and Best Practices

Now let's talk about editions and see what on-premise editions you can encounter when working with SQL Server. Enterprise and Standard are the two most common ones that you will see in production. Developer is now a free edition for testing-only purposes. Functionally, it's Enterprise equivalent. Express is a free, highly limited edition, while Web is a special one, which you will probably see less often. Why do we care about edition? The various editions have feature limitations by design that can impact maintenance efficiency as well. But the most common problem will be the scale and the resource utilization limits across the editions that cause performance- related problems, even unnoticed sometimes in production. Also, obviously, editions come with cost implications, Enterprise being the full-fledged, all-feature, most scalable edition, having the highest price, calculated per CPU core. Standard is cheaper but more limited. Developer and Express are free. How do I know which edition has what limitations? There is an official edition and feature matrix documentation always updated for each new major version, which should be reviewed in the solution planning phase. Please check the Bitly link above. An example of a major scale limit is the allowed number of CPU scores or the size of the buffer pool memory in each edition. It might be non-trivial to track down the implications caused by these limitations, and also to realize that your SQL Server doesn't utilize all resources thrown at it due to the edition. Similar to version, you can query the edition with T-SQL too, as shown in the above code examples. Let me highlight a few key points also as a summary for versions, editions, and patching.

It is absolutely a best practice to keep your major version and patch level supported, and with the major version, keep it in the mainstream support phase if possible. Do continuous and proactive patching with CUs and for older versions, SPs, too. Test new updates before going into production, and set up a retention and a patching policy in house for consistency. Use the Developer Edition for test purposes so that you can do load and scalability tests properly. When deciding with a major version an edition of SQL Server, verify the feature and edition matrix to prevent surprises.

Server Instances

In this section, I talk about on-premise SQL Server instances. So, what is an instance? The SQL Server database service is instance aware. When connecting to an on-premise SQL Server or a SQL Server environment running on a VM in Azure, we connect the server instance. Each instance has its own configuration, system databases, TCP port, and of course can host a set of user databases. Instances can be hosted side by side, and even major versions can be mixed. So I have an instance of SQL 2016 and another one which is SQL 2017, for example. What happens when I host multiple instances on the very same server? Let's say our host server, no matter if it's a physical or virtual server, has 8 CPU cores and 64 GB of RAM, some sort of underlying storage, and network capacity available. Our instances, 3 instances in our example, that will share all the available resources in the host server, that is, the 8 CPU cores, 64 GB of RAM, storage, and network bandwidth, and capacity will be shared or distributed among the 3 instances, plus among any of other services running on the host server and, of course, the underlying OS requires the resources as well. Each instance might have different performance requirements and characteristics, and each can have its own custom configuration. The instances can also run different workload patterns. One instance might run OLTP or transactional workload requiring single CPU core performance and the right optimized very fast storage. The other might run reporting or analytics workloads requiring lots of memory and multithread scalability across multiple CPU cores, and the third one, well, since this environment was provided by our customer maybe, we don't know anything about the third instance. That instance runs some third-party unknown workload. We are in a shared and mixed environment. How do we deal with multi-instance shared and mixed environments? Well, we don't prefer them, especially in production. Stacking SQL Server instances can cause scalability problems and make configuration, troubleshooting, and optimization more complex. As we need to take external factors into account too, anything that lives outside the server instance. Database administration tasks patching guaranteeing SLAs overall will be more problematic. Virtualization with single-instance deployments together with licensing optimization, a separate and highly complex topic in itself, can be a good server consolidation approach instead.

Server Configuration Options

In this section, I talk about server configuration options. We have seen that we deploy server instances on-premise. As noted, each instance has its own configuration and system databases. While speaking of server configuration, it then means instance-level configuration affecting everyone on the same instance. What can we configure at the server instance level? We can configure CPU, memory limits, global parallelism settings, and many other miscellaneous settings. These settings can be configured either via the UI in SSMS or scripted with T-SQL directly. When scripted, we can use the sp_configure system stored procedure and check the current set of configuration in the sys.configurations view. Out of the many available server configuration options, and this can vary with major versions too, of course, there are a couple that we will need to set in all cases from the default value after installing the server and definitely before going into production. Other settings might need to be changed, depending on workload characteristics. There are a few which should never, ever be set. The must-change configuration options consist of memory settings and many times, parallelism-related settings, which I will detail and demonstrate in later modules. Server configuration options must be verified, changed accordingly, and tracked in an ongoing manner. This is generally a part of a server health check process. There are other server instance-level settings and configurations that are set at different locations, like trace flags, tempdb. And a good place to have a quick review of a server setup is in the SQL ERRORLOG files.

Database Configuration Options

In this section, I talk about database configuration options. Databases also have their own configurations that can be set with different methods. The various database set options are available in all major versions and used to configure common database configuration options like compatibility level or recovery model, and there are dozens more. Beginning with SQL 2016, database scoped configurations were introduced. This, as we will see, is used to add an additional layer of settings that were previously only available at instance level or implemented by obscure trace flags, potentially increasing the configuration complexity. We set many common database properties with SET options, for example, the COMPATIBILITY_LEVEL and the recovery model. A server health check process should reveal any exotic or unexpected database SET options too. You can verify all these settings in the sys.databases view in all major versions. SQL 2017 introduced a new automatic tuning feature which has its own configuration in the sys.database_automatic_tuning_options view. You can also use the DATABASEPROPERTYEX() function to directly check for a particular attribute. I already noted the new database scoped configuration options introduced in SQL 2016. For example, previously the max degree of parallelism setting MAXDOP could only be set at server instance level that could be overridden at the single query level. Now if you want to override MAXDOP at the single database level while everyone else is using the instance level setting, you can do it with scoped configurations. Similarly, there are behavioral options here, like using the LEGACY CE or turning on OPTIMIZER_HOTFIXES that were controlled by trace flags only previously, making configurations more obscure. It's important to note that we now have a configuration hierarchy. It is very important to understand this when troubleshooting a problem or choosing a

solution, which configuration hierarchy level to verify for which settings and also in which layer we can apply our solution. Server and database-level configurations are better visible if you use query level hints. Those make troubleshooting harder. This code example shows how to configure MAXDOP at different levels, server, database, and query level. This code example shows how to use the new USE HINT query options in SQL 2016 and above.

Trace Flags

In this section, I talk about trace flags. So what are trace flags, and why do we care? A trace flag, or TF, is a switch that alters SQL Server behavior. As you work with very recent versions of SQL Server like SQL 2016 and above, the need of trace flags is somewhat reduced compared to previous versions, but still might be applied to different scopes. The use of TFs, especially if used in a mixed way at different scopes, can make troubleshooting harder, while they can make significant impact, pro and con. They are used in a format, TF and the number. For performance troubleshooting, a few well-known TFs used in previous versions have become de facto standards, made their way into the product from SQL 2016 and above out of the box. They can be set at server level, globally, at session level, or at single query level, as query traces on query hints. Out of the hundreds of TFs available in SQL Server, there are a few which have become common for performance optimization like TFs for tempdb optimization, other TFs for controlling automatic statistics update thresholds, for example. As noted, the need for using TFs in the old way has been reduced from SQL 2016 and above. The reason for this is that the few common TF behaviors have become by default behavior in the latest dbcomp levels, while others are encouraged to be set with database scoped configurations or the new query hint options instead. All these can make significant impact and differences. What are these TFs exactly? A few common TFs shown are TF 1117 and 1118 for tempdb optimization, 4199 to enable Query Optimizer hotfixes, 2371 to control automatic statistics update thresholds. Also 9481 and 2312 to control which CE version to use, the new or the legacy one, vice and versa different scopes. The letter has become important with legacy workloads migrated to newer SQL Server versions in order to use other new features in the latest dbcomp level, but also mitigating query plan regressions when using the new CE version. I talk about it in later modules. To add to the complexity, TFs can still be set in all versions. They either have an impact or not used by SQL Server, even if they are set. It's better to follow the major version's specific recommendations and documentation. If one plans to use a particular TF, maybe there are better alternatives, or it's not needed at all. This code example shows how to set a TF at different scopes with T-SQL. This code example shows how to set the TF behavior with SQL 2016 and above, specific query use hints, and database scoped configurations options instead. This code example shows how to verify which TFs are enabled at which scope and also how to turn a TF on and off. To sum up why TFs do matter is that they can control crucial behavioral patterns, pro and con, and potentially can be hard to detect their use, especially when used at different scopes in an undocumented way.

Tempdb

In this section, I talk about tempdb, which is one of the primary contributors to performance problems in on-premise SQL Server environments. Why do we care about tempdb? Because it's a common shared container in the very same instance our workload runs, used by everyone within that instance. Even if you don't really use it that much directly, SQL Server uses it under the covers. However, the common practice is that tempdb is used more often than not directly by applications and many times unfortunately abuse it without being aware of it. The most common use case is to create temporary user objects, local or global temporary tables by the application, that will be created in tempdb, as shown in this code example. What else is using tempdb then that we are concerned of? Well, besides our own user objects, which also can be table variables and table-valued function return tables, the version store lives in tempdb, for example. It is used for AFTER triggers, transaction isolation levels like read committed snapshot isolation and snapshot, and if you use Always On availability groups with read-only replicas, other internal structures like work tables and work files for sort operations, hash joins and aggregates, cursors, or use tempdb. If there are query plan problems, resource utilization problems, memory pressure scenarios, tempdb spills can also be expected, putting additional pressure on tempdb and cause query performance problems. There are two factors we really need to care about with tempdb configuration and performance optimization. I/O, which means that our tempdb data files and transaction log files should reside on a good performing storage, preferably on an enterprise-grade SSD storage that is proven to handle our I/O load, both read/write with varying I/O sizes. Internal allocation means that how SQL Server manages its metadata and internal allocations within tempdb is optimized. The letter is not that trivial, and our options might depend on what major version we are running with. I talk about this in more detail in a later module. Depending on which major version we are running with, we have different options for tempdb configuration. Earlier versions required all manual configurations. Recent versions help us with improved and built-in optimized behaviors as well. To sum up tempdb and performance requirements, we need to address native storage problems, memory bottlenecks, configuration problems, but let's not skip solution or application design either. If the application has been written in a bad way with the excessive usage of cursors, with the excessive usage of, in turn, temporary result sets, procedural instead of set-based query design, triggers unnecessarily, at one point you will need to rewrite your application so that it scales better.

Transaction Log and Recovery Models

In this section, I talk about database recovery models. There are three database recovery models in SQL Server, and you will encounter two the most often. The first one is simple, the transaction log is automatically cleared by SQL Server, thus log space can be reclaimed. We have the option to do full and differential backups only. There is no option for point-in-time recovery in this model. The chance that the transaction log size would grow unexpectedly is low, unless you have long-running or runaway transactions. The second one is full. The transaction log is cleared only when doing log backups. We can do full, differential, and log backups. Thus, we do have an option for point-in-time recovery. It is a common problem that the transaction log size grows unexpectedly, or otherwise, the log size

management is problematic in this model. This is a model which is also required when running the database in high availability or disaster recovery solutions like Always On availability groups, database mirroring, or log shipping. The reason I am discussing this is because the recovery model has impact on transaction log file management, and there are common myths around the relation of it on performance. I am not discussing the bulk-logged model separately in this module. In this section, I talk about the transaction log. The transaction log is a crucial component of a relational database implementing ACID properties. In SQL Server, everything is a transaction, implicit or explicit. By implementing write-ahead logging, the log provides a mechanism for database recovery in case the server crashes, for example, so that the transactions are either redone or undone or rolled forward or rolled back, making the database consistent. The corrupted or missing transaction log file having the extension of LDF can be disastrous for the integrity and consistency of our database. Also, writing the log efficiently and fast is overall very important for transaction performance. How does the transaction log file work? Let's see a high-level overview for now. Each database has one log file, or should have one and only one log file, which is an initial file size with the extension of LDF. Internally, SQL Server manages VLFs, or virtual log files, for writing the LSNs, or log sequence numbers, within. The general I/O pattern of the log is sequential write. We have four VLFs from 1 to 4 in our example colored with orange. Then we are writing the log; we let 5 and 6 get full too. What happens next? Remember that the log file could be reused if it is cleared. Also in full recovery model in order to clear the log, I have to do a log backup. In case I don't do that, or I have a long-running transaction filling up the log, the file size needs to grow physically. So in our example, if the log does not or can't clear, it grows with some growth size that is configurable at database level. This can happen a lot of times, making the log file size very big. When the log clears, SQL Server then can reuse the log. So the big question is how to manage the transaction log properly, which is often a challenge with production workloads. When running in the full recovery model, frequent enough log backups are mandatory, not just because of point-in-time recovery, but for file size management too. Instant file initialization does not help with growth optimization here. I talk about it in a later module. Each database file growth value must be changed from the default percentage growth to meaningful fixed growth rate, and also it's recommended to have the file presized. Try and prevent long-running transactions, like deleting millions of records all at once, as that can fill up the log quickly. Instead, do that in chunks, for example. Do not physically shrink the log file regularly because it will grow again, and we have seen that each growth event can be expensive. And of course, log file I/O, especially write performance, is critical for overall transaction performance.

Memory Management and SQLOS

In this section, I talk about the SQLOS and the resource management. SQL Server has its own resource management. It has its own scheduling, for example. Overall, it really does matter how the CPU architecture looks like under the SQL Server instance, how many and how fast CPU cores are available for my workloads. It depends on the whole system, but also on configuration and edition. Also, it allocates memory for many different reason, for

caching data pages, the query plan cache, connections, external components, or query execution, for example. I/O performance is highly important too. It writes the transaction log sequentially, reads and writes the database files with random I/O patterns with varying I/O sizes, also creates and restores backups. The component that incorporates operating system-like resource management functionality is called SQLOS, or SQL operating system. If you take away any of these resources without measurements and justification or don't provide enough resources to SQL Server in the first place, our workload performance and scalability will suffer. In this section, I talk about memory management. Out of the many different types of memory allocations used by SQL Server, the plan cache memory is one we deal with very often. What is it used for exactly? When executing a new query, whether as an ad hoc query or part of a stored procedure, for example, SQL Server will compile an execution or query plan for it. The plan will consist of physical plan operators that essentially will provide the means to carry out the requested query operation. In our example, we search for an order with ID = 1 in the Sales.Orders table to return its OrderDate column value. For this, SQL Server chose a Clustered Index Seek operator to locate that qualifying record. This plan, which was compiled first, is then stored in the plan cache memory. In case the very same query pattern is executed next time, it could be fetched from memory instead of compiled again. The plan cache memory is managed dynamically by SQL Server and is under throttling. The cached execution plans can be queried via the sys.dm_exec_cached_plans view to investigate what plans were executed. These are estimated plans though, and the plan cache is flushed when the server restarts, or we can flush it directly too. Our aim is to have a stable plan cache size, preferably with plans used multiple times; otherwise it can be bloated with many ad hoc plans that were used only once. Application design is crucial, like relying on objects like stored procedures instead of ad hoc queries to a larger degree, so that plan reuse is more efficient. When speaking of optimal workload performance, the most important memory allocation in SQL Server is the buffer pool memory. This is the memory where SQL Server caches the data pages direct from physical storage. The database files with the extension of MDF, and also there are secondary files with the extension of NDF, allocate data in 8K pages. SQL Server many times handles I/O of 8K pages, or 8 times 8K pages called extents. What happens when I first execute the query, the search for an order of ID = 1 in the Sales.Orders table. The relevant record resides on an 8K data page. SQL Server will see the requested data pages in the buffer pool memory already. If not, it reads the page in from physical storage, wherever and whatever that is. The pages are read into memory, and the query is satisfied with the data from memory. What happens when I update the very same page with another query? It all happens in memory again. As now the page to be updated is already in memory, we modify that and flag it as a dirty page. The update transaction is now hardened in the transaction log. At some point later, the changed page will be written back to the database file too. As you can see, it's vital to have a proper buffer pool size that is optimal for the particular workload patterns to reduce the need for physical I/O operations, as a physical read is still much slower than a read from memory, which is called logical read. The buffer pool memory size can grow and shrink. It grows when memory is available for allocation, and it needs to grow due to workload patterns to cache more data pages read from physical storage. Normally, we don't want it to shrink though, but if you don't configure the environment otherwise, and other external processes require more memory, SQL Server might need to

deallocate memory and flush data pages from memory to disk, which is again an unwanted scenario. If the buffer pool memory size is not optimal and can grow further and/or shrinks the due to external memory pressure, the I/O load increases due to having to read/write pages in between the buffer pool and the physical disk, causing serious workload performance problems and even server stability issues. I talk about buffer pool size management and configuration in more detail in later modules. It is crucial that our SQL Server instance has an optimal buffer pool size, depending on our workload patterns; otherwise I/O pressure increases and our workload performance suffers. What's in the buffer pool? We can query that via the sys.dm_os_buffer_descriptors view, which also comes in handy when trying to figure out which databases consume what size of buffer pool memory currently. I talk about the various server-level configuration options, especially about max and min server memory in detail later on. Memory management is of course a complex topic, and our configuration options are manifold too. I would recommend checking out KB2663912 on changes regarding memory management, beginning with SQL 2012 in relation to configuration options like max and min server memory. As noted, there are dozens of other memory allocations in SQL Server that also can be investigated in detail via views like sys.dm_os_memory_clerks and sys.dm_os_memory_objects.

Wait Statistics and the Threading Model

In this section, I talk about the threading model and wait statistics. So all in all as I highlighted earlier, our troubleshooting and optimization paths can be widespread and overwhelming. I don't even know where to start my investigation. Should I collect traces of system performance, collect query plans and runtime statistics, check up on maintenance jobs, some other settings in the environment maybe? It can all take an awful lot of time and effort without having the slightest idea where the root cause can be. Isn't there a method that helps me know where to begin and narrow down the problems? I would ask SQL Server where the bottlenecks are, and based on that response, I could drill down into the related layers for more details. Fortunately, there is, it's called wait statistics, and it's based on the threading model which SQL Server uses. SQL Server has its own scheduling. In a Windows OS environment, Windows uses the preemptive scheduling model. SQL Server, however, uses the non-preemptive, or sometimes it's called the cooperative model, which better suits a general-purpose relational database management system. Based on the available CPU cores, and remember, it depends on the host environment, configuration, and even the SQL Server edition, SQL Server uses objects called schedulers. So one CPU core means one scheduler in our model. And from now on, let's represent a scheduler with a three-piece component with a clockwise process flow which I discuss in more detail next. Similarly, if we have two CPU cores, we have two schedulers in our model. If we have four CPU cores, we have four schedulers in our model, and so on. You can check how many schedulers are available for a SQL Server instance in the sys.dm_os_schedulers view. Let me highlight another concept called parallelism here, which means a query plan operator like an index scan can be executed in multiple schedulers in parallel to improve performance and scalability. Now let's see how this three-piece component process flow works

with our scheduler, which is essentially the threading model. At any one time, only one worker thread can use the processor. That's when a thread is being executed and is in running status. This is a non-preemptive model, which means the worker thread voluntarily yields the processor in case it requires a resource or when it uses up its quantum, which is 4 ms in SQL Server. So our thread is in running status, and two things can happen. The first case, it requires some resource, whatever it is, waiting for a page to be read in from physical storage or lock resource, for example, and then it yields the processor and moves to the waiter list and gets into the suspended status. It stays on that waiter list until the resource it waits for gets available and the thread gets notified about it. Second case, it uses up its quantum and directly goes generally to the bottom of the runnable queue, which is a FIFO, first-in, first-out queue, and gets into the runnable status. There the threads are waiting to be executed on the processor to be in running status again. This is a clockwise cycle, which is fully tracked by SQL Server. The time it takes to get from running to running again is the total wait time. The signal wait time is the time waiting for the processor in the runnable queue. If we subtract signal wait time from the total wait time, we get the resource wait time, which is essentially how long a thread has been waiting for a resource to be available. All these wait times are measured in milliseconds. In SQL Server, we generally refer to SPIDs when executing workloads. In this model, let's assume SPID61 is using the scheduler, and SPID72 and 74 are on the waiter list. The good thing is that SQL Server assigns wait types on the waiter list, so we know that SPID72 is waiting for a wait type called WRITELOG, which is waiting for transaction log I/O operation to complete. SPID74 is waiting for a lock type wait, which is waiting for a lock resource. In the meantime, SPID84 and 86 are waiting in the runnable queue for the processor in a FIFO-type way. When SPID72 or 74 get notified in no specified order that their resources are available, they move to the runnable queue. To sum up, the waiter list is an unordered list where threads are waiting for a resource to be available with a wait type assigned. We also know the time spent here, which is the resource wait time. If you want to check how the waiter list looks like at any moment, you can do that in the sys.dm_os_waiting_tasks view. Wait statistics is the method that relies on the threading model I just showed. The sys.dm_os_wait_stats view is a server level view in SQL Server on-premise that shows what wait types have been occurring since the last restart of the server in a cumulative way. That is, everyone in the same instance contributes to the server level statistics, which then can be used to know what are the prevalent wait types and their maximum and average wait times of the server as a whole. In Azure SQL Database, these statistics, as we will see, are available at the database level, as there is no server instance there, or at the query level in the Query Store feature in SQL 2017. With custom extended event traces, we can also collect wait statistics at the session level too by providing a SPID as a parameter, for example. So essentially, wait statistics can answer the question, what are the performance bottlenecks in SQL Server as a whole in my database or in my particular query? Wait statistics also provide troubleshooting patterns, since the wait types, especially the most common ones, are well known and well documented by now. We know what the wait types actually mean and in what conditions in the SQL Server code they are used. If you find that the most prevalent wait type is WRITELOG with bad average wait times, then I know I have problems with transaction log I/O, which then requires specific troubleshooting. Maybe the storage where my log files reside can cope with sequential write I/O load.

PAGEIOLATCH_SH means SQL Server is waiting for physical page I/O reads, which can indicate both a slow storage under my data files, but it's more common that the server has buffer pool memory pressure due to memory sizing problems. Thus the problematic memory configuration results in subpar I/O performance. CXPACKET means that there are parallel workloads executing on the server, which can either be a good or bad thing; it depends. If I don't expect any parallel executions and it pops up, then I need to investigate where it's coming from. If it's occurring as expected but the wait counts, along with the average wait times, seems subpar and especially these occur together with PAGEIOLATCH_SH waits, then I can suspect that some workloads do large scans, which are done in parallel and read in unnecessarily large amounts of data maybe. Thus I can then suspect index usage problems, missing indexes or bad indexes or index maintenance problems with out-of-date statistics. At worst, I have query design problems that I can then troubleshoot that specifically. Sometimes a particular wait type helps me directly. Sometimes I correlate wait types with each other. All in all, wait statistics are my troubleshooting entry points. We ask SQL Server for the most prevalent wait types. We can do that in different scopes. Also we can collect that during a particular operation, if we want. There is a comprehensive wait type library with troubleshooting guidelines at the above URLs. If we want more efficient performance troubleshooting and optimization processes, wait statistics is the method that we should use. And that's a wrap-up for this module. I have gone through a few concepts that often come up when troubleshooting SQL Server performance problems. I talked about on-premise SQL Server versions, editions, and the importance of patching the server. I talked about server instances and what the performance and scalability impact of multi-instance environments can be. I talked about the various levels of configuration options at instance, database, and query level. I talked about trace flags. Then I showed why tempdb can be a common bottleneck in on-premise SQL Server environments. I talked about recovery models and the transaction log a little bit. And finally, I showed some memory management concepts, along with our primary method for performance troubleshooting and optimization called wait statistics, which is based on the threading model that SQL Server uses.

Optimizing SQL Server Instance and Memory Configuration

Server Health Check

Hello, and welcome back to my course, Managing SQL Server Database Performance. In this module, I guide you through a manual server health check process and discuss SQL Server instance-level configuration options, including memory and parallelism-related settings, among others. Checking up on your server configuration regularly is a best practice, and you can even prevent major problems from occurring in the first place. How does a SQL Server health check process look like? When you have many servers, dozens, or even hundreds that you need to manage, you probably want this process to be automated and just run it from time to time. You can, for example, implement your cost and data collectors in T-SQL, in PowerShell, or you can even customize your already deployed monitoring solution for this purpose, if possible. There are well-known free health check solutions available too that you can use. Many cases, you will need custom data collections because you might have components or configuration options that are specific to your application or solution. The health check analysis should pinpoint any deviations and changes from the desired configuration and can even provide an executive summary report. The key point is that the health check data collection should be lightweight, not impacting the production environment adversely, and you should persist the results into a database for historical analysis. The data collection should be timestamped, and storing the results in a repository also serves documenting and change-tracking purposes. It is asked many times, how often do we need to do a health check? Well, it depends. For example, you can have a frequently changing environment with high workload variations, and you also have a people with administrative privileges, and configuration might even change accidently sometimes. I definitely recommend a health check after a new installation of SQL Server to know the platform you start working with in detail to establish a baseline configuration or even discover unexpected configuration right at the beginning. Also, before going into production, as by then you should have an overview of system performance and behavior that might trigger an adjustment in configuration to better suit your production needs, and then regularly in an ongoing manner as part of IT and database operations also based on monitoring data analysis. When talking about the scope of a server health check, it generally includes the following: host environment configuration, hardware and operating system-specific information, CPU architecture, server power plan settings, for example; SQL Server configuration like configuration options, trace flags, tempdb configuration, versions, and patch levels; database settings like set options and scoped configurations; SQL ERRORLOG files, checking up on the server startup sequence for configuration overview, as

well as for any errors and exceptions; scheduled jobs, highly important to know if backups, integrity checks, and index maintenance are scheduled and run successfully, anything custom based on cache performance counters, DMVs like for server wait statistics, other specific exceptions, memory dumps on the server, and so on. The full server health check process collects lots of data. The data can be collected both from the host OS environment, also from within SQL Server. Couple of data collection method examples can be seen above, when we want to collect system-specific data about the underlying hardware, OS, if the server is crashing with memory dumps, for any exception patterns, and signs of database corruption. Without first running performance traces like a perfmon or extended event trace, SQL Server data collections like the above can also be used either as a snapshot or over a period of time to have an overview of the performance health and characteristics of the server instance. In both cases, the SQL error log files contain valuable information about the overall health of the server, and therefore it should be one of the first things to check when looking at SQL Server performance problems.

Demo: Memory Settings

In this demo, I'll show you how to configure a few important server configuration options, including SQL Server memory-related settings. SQL Server instance level memory settings are must-change server configuration options. That is, after installing a new server instance, these settings have a default value, which means the memory settings are not configured. We must set them and adjust the configuration later on based on measurements, if needed. No one knows the optimum memory configuration until measuring memory utilization, both at system and SQL Server level. I will primarily use Windows Performance Monitor counters, along with SQL Server DMV queries in a later module to showcase memory diagnostics. The optimal memory configuration depends on many factors, like the particular environment and workload characteristics. Now let's see a few scenarios how to approach the proper configuration here. I already talked about the buffer pool memory. It can grow and shrink. The buffer pool is the largest memory allocation in our SQL Server, and we want it to be sized properly to support our workloads. Essentially, we can control its size with max and min server memory. Max server memory is the maximum allowed size, while min server memory is the lower limit that we want to provide to our instance in case SQL Server is required to deallocate memory in case of external memory pressure. So when the buffer pool shrinks, we do not allow it to shrink below min server memory. Think about min server memory as protecting the instance from other processes. Scenario 1. In this scenario, we have a dedicated SQL Server machine, a single instance deployment. Remember this is our preferred scenario for production. A dedicated environment means that we have no third-party, custom, or other SQL Server-related services like SQL Reporting Services, hosted besides our single database instance that would have significant memory footprint. We have 16 GB of physical memory, or RAM, available for our Windows server here. I must then set max server memory accordingly. The primary approach here is to leave a few gigabytes of memory to the underlying OS. Leaving 4 GB is a good start. That means I can set max server memory to about 12 GB. If this is the optimal setting, I will only know it from diagnostics data by measuring the environment. I

might be able to increase the setting a little bit, or I will need to decrease it, or it can be that all together, having about 12 GB of max server memory is simply not enough for my future workload. And then I need to upscale the server, add more memory to it, and adjust max server memory again accordingly. Scenario 2. In this scenario, I have some custom services hosted on the server, besides my SQL Server instance, having additional memory footprint. Thus, I need to reduce my database instance max server memory to leave 4 GB to the OS and leave additional memory to those services. If I know the extra memory footprint by the numbers, that's good. If not, we first reduce max server memory and then measure memory utilization and adjust the configuration accordingly later on. In this case, I set max server memory to about 10 GB. Thus, I left an additional 2 GB of memory to everyone else. Scenario 3. Now let's have an example when we have a multi-instance shared environment. I have a SQL 2017 and another SQL 2016 instance on the very same server. I still have 16 GB of RAM, so this will need to be shared and distributed between the 2 instances. I leave 4 GB to the OS, and then the remaining 12 GB will need to be split between the 2 instances. If I know the proportions in advance, that's good. If not, I can simply split 12 GB in 2 and set both max server memory settings to 6 GB for a start. What about min server memory? This is used a guard against other processes. Setting this in all scenarios is recommended, but in a multi-instance environment or in a server with other services, setting this is a must too. In the 2-instances scenario, we can set min server memory to 2 GB, for example, for each instance, if we know that our instances can live with 2 GB, in the worst case. Min server memory has another role in virtualized environments like in VMware. A VM can be set up with reserved memory to provide fixed allocated memory to the guest VM. VMware administrators do not like this at all, or reserve only a small amount of memory. For example, we have a 32-GB VM, but that 32 GB is not available all the time, and VMware might even take memory away, if needed. SQL Server doesn't really like scenarios like this, so in case we set max server memory to 28 GB, what happens when we are left with our VM reserved memory only? For that scenario, setting min server memory to a value below the VM reserved memory is recommended. Let's see it as an example. Reserved memory is 14 GB. At worst case, we still need to leave about 4 GB to the OS, so setting min server memory to 10 GB is a good start. Lock pages in memory. Depending on many different factors, even OS versions, when external memory pressure scenarios happen, the OS might page out a SQL Server process memory, essentially the buffer pool memory that we are concerned of, to disk, causing serious performance degradation for our workloads. This is an event that is logged to the SQL error log files too. In order to prevent such scenarios, we can set the lock pages in memory policy setting by adding the SQL Server database service account to the security policy. The best practice here is to only configure this when this problem can occur or has already occurred in your particular environment. Optimize for ad hoc workloads. There is another setting that relates to memory configuration. It's connected to the plan cache memory in particular. When your database solution uses ad hoc workloads primarily, it can be that the plan cache memory will be full of plans that were used just once and then not really reused. In this case is the plan cache memory set to be bloated with single-use ad hoc plans, and this can be in the few gigabytes range, which might be a problem with servers having smaller amount of memory or otherwise have memory pressure problems. When your application has this sort of workload characteristics, the optimize for ad hoc workload server configuration

option is recommended to enable it, to set it to 1 from the default 0 value. With this setting enabled, when an ad hoc plan is being compiled and put into the plan cache for the first time, SQL Server will not store the entire plan, only a small plan stub. Next time when the same plan would be used again, the compiled full plan is then stored and the stub is removed from the plan cache. This mechanism results in a much smaller plan cache memory requirement for truly single-use ad hoc plans.

Demo: Parallelism Settings

In this demo, I'll show you and talk about parallelism-related settings. MAXDOP. Max degree of parallelism, or MAXDOP in short, is a setting which can be controlled at different scopes within a configuration hierarchy. We can set it at server instance level from SQL 2016 and above as a database scoped configuration, and also at query statement level with query hints. At each level, we can override the higher-level setting. When installing SQL Server, MAXDOP has a default value of 0. It impacts parallel execution plans. It means that all schedulers can be used for parallel executions, and this might not be the most optimal setting. Parallelism is generally a good thing, especially with read-heavy workloads against larger volume of data, where larger index scans or seeks are required, for example, with reporting workloads. Parallelism, when executing a query plan operator across multiple schedulers in parallel can improve performance and provides the multi-core scalability in SQL Server. The query optimizer might decide with a parallel plan based on different factors. Of course, query cost is the most prevalent factor, the optimizer being a cost-based optimizer. So while expected and planned, parallelism can be beneficial for overall query performance. Unexpected or excessive parallelism might cause problems due to bad queries, bad or missing indexes, or out-of-date statistics. It's a good practice to change the server instance level or MAXDOP setting to a fixed value according to the actual environment in advance, then again monitor the performance and adjust if necessary. There is no absolute good value here. The actual setting depends on the underlying hardware, even SQL Server versions, and of course your workload patterns. There are products out there that have a fixed prerequisite of MAXDOP = 1, which means that parallelism is turned off. If you cohost databases for those products with custom databases that otherwise would benefit from parallelism, you can choose to separate the databases onto different server instances, or from SQL 2016 onwards you can override the instance level global MAXDOP setting with database scoped configuration too. When detecting excessive parallelism that might cause process problems, CPU issues, for example, do not turn MAXDOP off, thus set it to 1. Instead, try optimizing your workload, check up on index usage, and adjust the MAXDOP to a more meaningful value. There is official guidance on how to preconfigure MAXDOP, which should be reviewed in all cases separately. Cost threshold for parallelism. Besides setting MAXDOP, you can also increase the setting from the default 5 to a higher value in case you do not want parallelism to kick in at lower query cost ranges, especially in case of excessive parallelism already causing problems. Whichever approach works best, or both, is really something that one needs to try in the actual environment with the actual workloads and monitor the impact of these changes.

Demo: ERRORLOG Files

In this demo, I talk about why it's crucial to look into the SQL error log files on a regular basis. The SQL Server error logs are one of the first things to check when doing a server health check or troubleshooting performance problems. The log files reside in the instance log folder, and by default, there are seven of them in total. There is one that which is the current and contains very recent log entries since the last restart of the instance. Its name's simply ERRORLOG. These are standard text files that you can open and analyze in any text editor. Six other log files named ERRORLOG.1 to ERRORLOG.6 are retained for historical analysis. You can modify how many such files you want to retain. Increasing the number of files might come in handy when troubleshooting intermittent problems and also when recycling the logs is frequent. The logs can be recycled with each service restart or directly with the system stored procedure, and recycling means that a new log file is started. The error logs contain a lot of valuable information about the health of the server, as well as can contain key performance- related entries. These are really straightforward, but many times simply missed, causing the troubleshooting process to be much longer. When reading the error logs, we can just simply start at the very beginning and get basic information about versions, the hardware and OS platform server name, for example. When getting used to what's in the logs, we can then search for where known patterns directed to. Also, it is an easy way to find obscure trace flags that were configured as startup parameters that might impact your code behavior and performance. We can see the CPU architecture along with any limitation coming with the SQL Server edition. I talk about Instant File Initialization for database file growth performance optimization later on. Beginning with SQL 2016, if this setting is enabled or not, it's logged in here too. The database startup sequence also shows what databases are there within the instance. We can see if any server configuration option has been changed, what was the old value, and what's the new value. In case there is serious memory pressure, the well-known memory paging exception is logged into the error logs, which really is a red flag before diving into further and detailed troubleshooting. If the server is crashing and creating memory dumps, mostly due to product bugs or external components loaded into the SQL Server process, the error logs show data about the dump creation and stack information. The dump files, by default, are created in the very same folder where the error logs are, by the way. Both the memory pressure and the memory dump entries are server stability red flags that require immediate attention. If there are serious I/O problems and the well-known 15-second I/O warning entries are logged into the error logs indicating which database file I/O has such problems, you can know that the underlying storage performance requires a closer look, and you might also need to involve the storage team if applies. Similarly, when database file autogrowth is a serious bottleneck due to storage problems or due to bad configuration, those problematic autogrow events are logged in here too. The error logs are also a very good source for all sorts of system-level errors and exceptions. Well-known error numbers like 823, 824, and 825 are logged when I/O operations did not succeed or where we tried, often indicating lower-level I/O or storage problems. When we really want to check up on the database integrity with DBCC CHECKDB to verify if there is corruption and also involve the storage team for further investigation. All in all, many times the SQL error logs directly show what the major problems

are at server level, and as we are moving from top down with our configuration and troubleshooting approach, checking up on the error logs should not be missed. And this is the wrap up for our module. I talked about the importance of regular SQL Server health checks, I showed how to configure SQL Server memory settings and a few other server configuration options including max degree of parallelism, and finally, I pointed out why checking up on SQL error logs files regularly is of importance.

Optimizing Tempdb and User Database File Configuration

Database File IO

Hello and welcome back to my course, Managing SQL Server Database Performance. In this module, I continue our server health check and focus on tempdb and user database file configuration options. Tempdb continues to be a top performance bottleneck with on-premise and Azure VM SQL Server deployments, so the proper configuration is of paramount importance. I also talk about a few other factors that can impact I/O performance adversely, along with how to monitor I/O performance in a SQL Server environment. A SQL Server health check data collection must include the following at least for tempdb and database file configuration. How many data files does tempdb have? How are those data files sized? Are they presized properly, and are they sized equally? There are two trace flags that need to be verified, too, but you shouldn't set them in all major versions. Before SQL Server 2016, TF 1118 and TF 1117 are recommended to enable globally, preferably as a -T startup parameter. Beginning with SQL Server 2016, these TF behaviors are built in, so these TFs should not be set anymore. They're deprecated, actually, and if you set them accidently, a warning is logged into the SQL Server error log files. Database instant file initialization has a positive impact on all database creation, autogrow, and restore operations for the data files only on the entire server instance, so it needs to be verified if it's on or off. File autogrowth settings for both data and log files need to be optimized in advance, especially for all user databases, tempdb, and many times for other system databases like MSDB2, as the default setting of 10% growth is far from optimal. When optimizing database file configuration and tempdb, there are other factors too that can have a major impact on performance, and these are very common in production environments. Troubleshooting these can be painful, as many times it requires storage or other server administrator skills. Thus you may need to involve other teams during the process. One of the most important factors, obviously, is the type of storage underneath your storage. If it's a traditional or SSD storage, how it's configured, is it tiered like in Azure that discuss in a separate module? How does the physical file layout look like? Have you separated the data, log, and tempdb files onto different drives or disks? This can have a positive impact on I/O performance, but it also depends on the type of storage, storage costs that you can allow, sometimes what makes sense from server administrator perspective. Drive formatting under the SQL Server database files, but also under backups, obviously before going into production can make a difference. The default 4 KB Windows setting is not the best for SQL Server I/O patterns, so using a 64 KB NTFS allocation unit size is generally recommended instead. One of the most overlooked areas is proper antivirus configuration. This is often invisible to DBAs but can impact I/O performance in a negative way. How to configure an AV software together with SQL Server is documented in KB309422, which should be reviewed and applied before going into production. Good I/O performance is crucial in a

SQL Server environment. In order to achieve good I/O performance, the underlying storage subsystem must provide good enough performance natively, but there are many other factors too. Anything in the I/O subsystem that can impact I/O performance adversely, AV software, SQL Server memory configuration, is there memory pressure causing an I/O pressure as a result, and how optimum my workload is? How do we measure I/O performance? There are various metrics like IOPS, throughput, and it depends what metric we prioritize with our particular workload. But in general, I/O latency is one of the most important metrics that we deal with. It tells how fast an I/O operation is completed from SQL Server perspective, whether it's a read or write, does it directly impact transaction performance? It is measured in milliseconds, and latency should be as low as possible on average. An old recommendation is that data file random I/O latencies should be well below 25 ms, while transaction log sequential write I/O latencies should be below 5 ms. These are still a challenge in today's production environments in general for multiple reasons. In order to improve I/O performance from a storage perspective natively, it is recommended to use enterprise-grade SSD storage under your SQL Server environment in production. Migrating SQL Server workloads to modern platforms with all-flash or SSD storage can improve overall workload performance significantly without changing anything else. However, this is not always true or can only work to a certain degree. To provide scalability and achieve low I/O latency ranges consistently, it still matters how the underlying storage is configured and if performance is tiered, if SQL Server memory configuration is optimal and there is no memory pressure causing increased I/O load, if the database application has been written based on best practices and workload is optimal, if physical database schema including indexing is optimal to provide optimal access patterns to the database engine, if other external factor, third-party components, AV configuration, or parallel I/O load in a shared and mixed environment do not interfere or impact I/O performance. Monitoring I/O performance in a SQL Server environment can be done with different tools and methods, but one of the primary methods is using Perfmon, or Windows Performance Monitor, with the LogicalDisk object counters to measure I/O performance over a period of time, what the latencies are, disk queue length, throughput, and so on. This gives an overall picture of I/O performance under the entire server environment. Within the SQL Server instance, SQL Server also tracks database-level I/O statistics, latencies, written and read bytes since the last restart of the instance in a view called sys.dm_io_virtual_file_stats. This should also be collected in a health check, see how I/O performance looks like on average from SQL Server perspective and which database files on which drives may suffer from subpar I/O performance. In case of nonoptimal values here, since these are averages over a potentially longer period of time, you can then run perfmon to trace the performance as it happens. Server wait statistics also gives an overall impression of I/O performance with well-known wait types PAGEIOLATCH_SH, PAGEIOLATCH_EX, AND WRITELOG. As I noted, subpar I/O performance might have different root causes; it's not always the storage itself that has problems.

Instant File Initialization and Tempdb Bottlenecks

I highlighted database instant file initialization as a setting that can make a positive impact at instance level on SQL Server I/O performance. Let's first see how file initialization works by default in SQL Server. All I/O operations are done via the OS and the storage layer. When claiming disk space for creating a new database file, when autogrowing a database file, or when restoring a database from backup, SQL Server refers 0 the disk space before it starts using it for writing data to it. This zeroing phase, or zero initialization, can take a significant time, depending on storage performance and size of the disk space to zeroed. Therefore, it can adversely impact file creation, autogrow, and database restore performance. Database instant file initialization is a setting that makes SQL Server skip zeroing the claimed disk space, thus skip zero initialization when creating a new data file, autogrowing a data file, or restore a data file from backup. A key point is that it applies to data files only. Log files will always be zeroed, no matter this setting is on or off. This setting is not a SQL Server configuration option per se; it is a security policy called perform volume maintenance tasks, so you need to add the database service account to this policy and restart the service to take effect. You might wonder why it's not on by default. It has a security consideration that you need to evaluate in your particular environment, but in general, the performance gains outweigh the risks, so turning this on should be considered. In a previous module, I highlighted the two most common problems with tempdb. The first one is a I/O bottleneck. When the tempdb files, both the data and log files, reside in a slow storage, for example, and we measure subpar I/O latencies for tempdb read and write I/O operations. The second is the so-called tempdb contention, which does not have anything to do with I/O performance; it's about internal allocation contention with in-memory metadata, and it can mostly be tracked with wait statistics. How can you see which problem or problems do you have? The I/O problems are more straightforward. Still many times it is unnoticed in production. Perfmon and the sys.dm_io_virtual_file_stats view provide real-time or average summary of I/O performance. Tempdb contention shows up as special wait types of PAGELATCH_UP OR PAGELATCH_EX with wait resources pointing to tempdb allocation objects. If these become prevalent with bad average wait times, addressing the contention problems is required. How to alleviate tempdb I/O bottlenecks? Well, the first and most obvious solution is to put the tempdb files on a better storage. Even if SQL Server uses a standard storage for its user database files and it proves to be okay, in case the solution is tempdb heavy, overall performance might greatly benefit from having tempdb on its own SSD storage. Using multiple tempdb data files addresses I/O problems too by having parallel I/O capability, but it's not required to have the multiple data files on separate disk volumes. Simply separating all tempdb files from other database files is mostly enough. It can be that further separating the tempdb log file from the tempdb data files is still needed. Having all these in place, further optimizing the workload is always welcome, and it can greatly reduce the tempdb pressure by alleviating unnecessary creation of temporary objects and loading large amounts of data in them or getting rid of cursors, for example. Addressing server-wide memory pressure problems can also reduce I/O pressure on tempdb. How to alleviate tempdb contention? If you use SQL 2016 or above, you are good to go already due to the built-in improved algorithms regarding tempdb allocations, and also the well-known TF 1118 and TF 1117 behaviors are built in. Also, the installer helps you to preconfigure the tempdb files already in the installation phase, way before going into production. Other than that, pre-SQL 2016 instances are recommended to set these TFs. In all

major versions, having multiple tempdb data files presized and equally sized with fixed and equal autogrow settings are all required to address tempdb contention problems. At worst case, the workload might also need to be optimized.

Demo: Autogrowth Settings

In this demo, I show you how to verify if database instant file initialization is configured, and talk about the file autogrowth settings too. Database instant file initialization. Let me show you a couple of ways to check up on whether instant file initialization is on or off. As this is a security policy setting, it might not be obvious in older versions. However, in SQL 2016 and above, if this setting is enabled, it's logged into the SQL error log files. Also, there's a view named sys.dm_server_services that you can use with older and patched-up instances. There is this column named instant_file_initialization_enabled, with a yes or no value. Other than that, you can check up on the security policy settings and verify directly if the database service account is added to the perform volume maintenance tasks policy or not. Database file growth settings. Database autogrowth configuration problems are common in production environments. Unfortunately, the default growth value of 10% is a bad setting, and it should be reconfigured before going live with your workloads. In order to check up on all database size and growth settings, you can use the sys.master -files view at server level and check the size, max_size, growth, and is_percent_growth column values. If is_percent_growth has a value of 1, it means that you definitely need to configure a fixed growth value, mostly in the megabytes range instead for that particular file, no matter if it's a data or log file. If is_percent_growth is 0, then the growth column value is the actual growth size, which you need to decipher as growth value times 8 in kilobytes, like 8, 192 times 8, which is 64 MB in this example. Similarly, you can check these settings in the context of the particular database with the sys.database_files view. Configure such growth values that perform well with the underlying storage and count with zero initialization if instant file initialization is disabled for some reason. If instant file initialization is enabled, you can specify larger growth values for the data files and smaller values for the log files, as instant file initialization does not apply to logs.

Demo: Tempdb Configuration

In this demo, I show you how to specify the number of tempdb data files and also talk about data file sizing. Tempdb trace flags, talking about TF 1118 and TF 1117 for tempdb, I noted that these are not required in SQL 2016 and above anymore. Still, if you set them for some reason, like here as startup parameters, warning messages are logged into the SQL error log files. Tempdb configuration. If you use an older version of SQL Server, you are on your own with tempdb optimization. By default, you will have one data file with 10% growth settings, and the trace flags are not enabled. You need to add more tempdb data files and configure tempdb and trace flags manually, depending on the actual environment. Beginning with SQL 2016, the installer helps a little bit regarding how many and how large

tempdb data files you want for a start. The number of data files depends on your environment, and you may even need to experiment with the most optimal configuration. However, the number of logical processors is used primarily to determine the starting number of data files, and Microsoft has an official recommendation on this. An installer uses this information to recommend the configuration, but even with this, you will need to further configure and optimize file sizes and growth values. In this case, I have a server with 8 schedulers available for my instance, and the installer recommended 8 tempdb data files, each 8 MB in size with 64 MB growth settings. If this configuration went live, I would definitely increase the data file sizes to a much larger value because I want my files to be presized as much as I can, and despite the 64 MB growth value is better than a 10% growth rate, I would increase the growth size to a larger value and of course having instant file initialization on. The key points are having multiple tempdb data files according to the number of logical processors available and follow the official recommendations here. Resize the files to a large value that prevents file growth, at least initially, and set a fixed meaningful growth value that is equal on all data files. And this is the wrap up for our module. I talked about the various factors that impact database file and tempdb I/O performance. I also talked about database file configuration and tempdb optimization options. I explained the difference between the two major types of problems affecting tempdb and discussed how to address them.

Configuring SQL Server in Azure

SQL Server in Azure

Hello and welcome back to my course, Managing SQL Server Database Performance. In this module, I discuss two major cloud SQL Server offerings in Microsoft Azure, SQL Server on virtual machines as an infrastructure as a service and Azure SQL Database as a platform-as-a-service solution. I explain the key differences and focus on what to watch out for in terms of performance and scalability when migrating Existing, or planning to deploy new solutions to any of these platforms. Companies often have a challenge with migrating existing workloads in SQL Server environments to the cloud. Similarly, while planning a brand-new database solution from scratch has its benefits, it is a similar headache to choose between the various and ever-changing cloud database offerings too. The major problems are estimating the costs and also correctly sizing the cloud solutions for performance and scalability while being cost effective. The number of existing database environments can be overwhelming without clear health check, performance baseline, and workload characteristics or workload pattern information about them. So many time, a cloud database solution ends up in a try-and-see scenario. Talking about a cloud platform like Microsoft Azure, the main drivers for hosting SQL Server environments are cost, this can be quite complex to even estimate, and depends on existing licensing terms too. However, it is often a clear target to reduce overall licensing or operational costs when running SQL Server in Azure. Operations and maintenance is generally a big cost factor with on-premise environments, especially when the number of environments also comes with varying complexity. Security and availability requirements. I discussed SLAs, or service-level agreements. Adhering to SLAs while also adhering to corporate policies is often a huge challenge with on-premise deployment. Availability in the SQL Server world comes with added technology and operational complexity in multiple layers, OS, network, storage, and within SQL Server itself. And also it comes with increased SQL Server licensing costs, so corporations often try to work around the costly solutions whenever they can. Built-in high availability solutions in Azure can be a main driver for Azure implementations. Modernization applies to all the legacy on-premise environments. We're constantly trying to keep them more up to date, but more often than not, on-premise SQL Server environments lag behind, and it either causes security risks or hinders performance and scalability. When evaluating the benefits of Azure database solutions, the main topics are cost and operations. However, the following advantages are less often discussed. Security. Azure as a cloud platform provides many integrated security solutions that simply do not exist with on-premise environments. Also the SQL Server Azure offerings provide built-in security features that might only come with the Enterprise edition of SQL Server, thus not always utilized, for example, Transparent Data Encryption, or TDE. Scalability is provided in various performance tiers, as the SQL Server solutions are all tiered in Azure. This is

again less understood and can easily become a problem when trying to estimate cost and size the solutions in advance. Of course similar infrastructure scalability on-premise that is provided by the Azure platform can be hardly matched, or with very high costs only. A clear advantage is the ability to upscale or downscale resources, depending on usage. There are solutions where scaling out is also an option. Some of the Azure SQL Server solutions, especially Azure SQL Database, has built-in automatic tuning functions that do not exist on premise at all, currently. Similarly, the Azure platform-provided performance monitoring and insight capabilities might not exist on premise due to the lack of proper enterprise and SQL Server-aware monitoring solutions. Now let's see what are the SQL Server Azure offerings currently. SQL Server on VM, that is SQL Server on virtual machines, is what we call infrastructure as a service. This is the most compatible with existing on-premise SQL Server deployments because these are the exact same SQL Server instances that you need to manage on premise, whether or on physical or virtual servers. You build up the underlying virtual servers, virtual networks, configure storage, and install full-fledged SQL Server instances that you do operations for, including deployment, batching, and all the configurations that I discussed so far. Azure SQL Database and managed instances, out of which I discuss Azure SQL database within this course in more detail, are platform-as-a-service offerings. Both are managed services, thus managed by Microsoft, but provide different capabilities and compatibility with your well-known SQL Server environments. Azure SQL Database is also called database as a service. You get separate managed databases, and there is no physical server instance available, while managed instances provide instance-level access and resemble more closely the full-fledged server instance. Still there are differences that you need to consider. All these offerings in Azure are tiered, so you need to count with costs, services tiers, and sizes in advance.

SQL Server Virtual Machines

I now explain SQL Server on VMs in more detail. As I noted, these are the very same full-fledged SQL Server instances that you have on premise already. As a result, everything that I discussed so far, all the following, are of importance when addressing performance-related problems. Planning and sizing the environment, deployment of SQL Server components, patching up the entire server environment, configuration both at server and database level, performance optimization approaches, Operations, and maintenance. It's all up to you. In an infrastructure-as-a-service model, all resources are tiered, including network components, and of course the virtual machines themselves. There are multiple VM series and sizes you can choose from, and the series different CPU architectures, as well sample in what sizes of virtual machines with what scale limits are available. A VM series comes with an Azure compute unit, or ACU number, which essentially tells you the score of computing power of that VM series that you can also use for comparison. Very important and many times less understood aspect is that all storage options are tier 2, including the temporary disk of drive D, which is provided with every VM. So when choosing SQL Server on VM, one of the first tasks is to identify the target VM series and the exact size within that series. Now let's see why VM series is important. It determines the available VM sizes, number of vCPU, and

memory configurations available, whether standard or premium storage is supported with that series. With SQL Server in production, only premium storage should be an option. The CPU architecture and corresponding ACU numbers. When a VM series is chosen, you need to choose a proper size, but this should not just be based on vCPU and memory; you need to evaluate temporary disk space and performance, maximum number of data disks, storage and network performance corresponding to that VM size, including storage IOPS and throughput and network bandwidth. Okay, but what is the best virtual machine choice for SQL Server workloads? Of course, it all depends what your workload characteristics look like. And once you move to a VM, you can resize the VM later on too, or you can deploy a brand-new VM in a new series and migrate to databases there. But it all comes with administration overhead, downtime, and the process itself can take some time, so it's better to estimate the size in advance. Do an Azure VM server and SQL Server health check before going into production and monitor performance. There is a comprehensive Azure VM catalog available and updated from time to time at the above Bitly link that you can choose from. Once you understand your workload, if you're relying on sheer single-CPU performance or require multi-core scalability or other larger memory size, you can evaluate the various VM series optimized for different tasks. But what is equally is important is to also review the SQL Server VM best practices documentation at the other Bitly link above. That includes SQL Server and Azure-specific configuration best practices, especially regarding storage tiers, storage caching, and tempdb configuration specific to Azure VMs. As a summary you don't just migrate based on vCPU and memory as they exist on premise; you have to count with further Azure-specific tiered resources and configuration too. Essentially, you need to do a regular SQL Server health check that gets enhanced with Azure platform-specific checklist items, including storage types and tiers, number of disks added, for example, to use P30 or above disks, storage caching settings per SQL Server physical file layout. For example, it is advised to use read-only caching on data and tempdb disks and use no caching transaction log disks. Tempdb layout if local temporary storage is used. For example, depending on VM size, write-heavy tempdb workload patterns can benefit from the local SSD drive D instead of an added data disk. But it's really something to try and measure. As a summary, SQL Server on VM performance factors that you need to manage and consider when configuring troubleshooting or optimizing for performance are the following: VM types and VM sizes, storage types and tiers, network configuration, additional cloud services used, SQL Server configuration, database configurations, workload optimization and patterns, patching and maintenance. Regardless of all this, SQL Server on VMs allows a relatively easy migration path for existing on-premise environments.

Demo: Creating SQL Server Virtual Machines and Sizing

In this demo, I create a new SQL Server VM in Azure, discuss VM sizing issues, Azure VM features, and SQL Server VM best practices. Let's check up on the Azure VM catalog and see how the categories look like. You can see the categories, for example, the general purpose VMs, which provide sizes that are good candidates for generic-purpose SQL Server workloads too. There are other categories like memory optimized or storage optimized that might also be

considered for SQL Server workloads, depending on your workload patterns. You can also check the ACU numbers for each VM series, too, for comparison. Now let's see what's in the general purpose category, what the VM series and sizes are available in there that I can choose for my SQL Server workloads. When choosing a VM size for SQL Server, in case you don't have any other special requirements for storage and memory, consider the DSv2 series first. As you can see, Premium storage is supported in this series, which is a must for production SQL Server deployments. ACU ranges from 210 to 250. For smaller SQL Server VMs, DS3 v2 or DS4 v2 sizes look good. See the differences in the temp storage sizes, and that indicates temp storage performance differences too. Also have a look at the differences and scale limits for storage throughput, differentiated with cached and uncached metrics, also network bandwidth. Now if you choose DS2 v2, but 14 GB of VM memory is not enough, and many times that's not that much as we for SQL Server, check up on other series too where we have more memory with the same vCPU size, but overall you might get less scalability for storage performance. Counting with vCPU size also matters of course for SQL Server licensing. Choosing from the memory optimized category, have a look at the DSv2 11 to 15 series. Same vCPU sizes come with much higher VM memory and comparable storage performance. Checking the estimated prices in the Azure price calculator, DS4 v2 with 8 vCPU and 28 GB of RAM has a similar price compared to DS13 v2 with 8 vCPU and 56 GB of VM RAM. For demo purposes, I create a SQL Server 2017 Developer edition VM, which is a free edition, and select a template from the Azure Marketplace. So I want the VM preinstalled by Microsoft. I often encounter such VMs even in production, of course with a non-free SQL instance, people assuming that these VMs are fully configured and ready for optimized production use as is. This is not the case, actually, and even your own policies might override configurations like preferred disk layout under SQL Server. In any case, I can't emphasize it enough. Do platform and SQL Server health check before going into production. A template selected from the marketplace might not be configured exactly how it should be, and there were even deployment bugs with a few of them, which problems can only be identified in a health check or worse, when you already suffer from production issues. Now let's see, what can I configure in advance for my VM? Selecting the size is one of the most important, as I explained. For now, let me choose a different size here, a DS3 v2 instead with 4 vCPUs and 14 GB of RAM. Let's again check up on the VM catalog what scale limits this size comes with exactly. For the disk configuration, I select Premium SSD for the OS disk 2, and of course make sure that you use managed disks only. I revisit this configuration later on, once my VM has been deployed. Let me highlight a few interesting VM features like automatic shutdown and the ability to add all sorts of extensions like anti-malware components and so on. I can also configure a few things for my SQL Server instance and make note that the data disk of 5000 IOPS and 1 TB in size will be added, which is a P30 SSD disk. You can select disk optimization options, but in any case to prevent surprises and template bugs, make sure that no matter what, you check up on the storage configuration per the SQL Server Azure best practices after deployment of the VM. Another feature is automatic patching, which is a nice feature. You must decide if this adheres to your operations policy. But you must think of patching yourself, as this is infrastructure as a service. Similarly, you can configure automatic backups. Now I create my new preconfigured VM. Once the VM has been deployed, let me check up on what extensions are installed with it. In this case, the SqlIaasExtension

component has been installed. I now check up on the SQL Server configuration for active settings and any features like Automated Backup configuration that I want to enable at the VM level. What I am more interested in for now is the disk configuration. As expected, I have 1 P30 data disk added in the deployment, and by clicking on it, I can check up on disk performance metrics already. Let's check the P30 disk prices in the disk pricing list. As I noted, if you want more disk performance, then you either choose a higher disk tier, P40 or P50 and so on, or configure striping, for example. I'm now connected to the VM with Remote Desktop and now check up on the disk layout. I have a system drive, drive C, where unfortunately SQL Server was installed too. This is an example that the physical deployment from a template might not be the most optimal. All the log files, log folders, system traces are hosted on drive C too. Company policies might require separating the OS and applications like SQL Server to different disks, actually. Drive D is the local SSD temporary disk, which you might need to use for tempdb, if applies. Drive F is the P30 managed disk used currently for SQL Server database files and tempdb. I always check up on system configuration with msinfo32, which is simple and straightforward to see CPU architecture and system memory available. At this point, I would highly recommend using nice utilities like CPU-Z to check up on detailed CPU architecture, properties, and performance. For completeness, checking up on VM power plan settings show that it is set to high performance, as I expect for a SQL Server VM. It's a best practice to format the SQL Server disks with 64 KB NTFS allocation unit size, so let's have a look with the command line fsinfo utility. Drive F hosting the SQL Server Database files including tempdb is correctly formatted with 64 KB NTFS allocation unit size. Drive C is used with the default 4 KB, which is okay in this case. Checking up on the SQL Server error logs, it shows that Instant File Initialization and lock pages in memory have not been configured, so I now go ahead and configure these policy settings. Similarly, I configure a max server memory of 10 GB and a MAXDOP of 2 for now. I don't like the fact that we only have 1 P30 data disk and the SQL Server data and log files, plus the tempdb files are not separated. I now assume that my application is not tempdb heavy, so I leave the tempdb data files on drive F and add the new P30 disk for hosting the transaction log files, including the tempdb transaction log file too, which is currently on drive F. Disk caching must be verified. The disk for data files and tempdb should have read-only caching enabled, while the new disk I add for logs should not have any caching enabled. Make sure that disk caching configuration is done when the VM is stopped and the disk is not in use. Once my new disk has been added, I make sure that it's being used by the OS and formatted with 64 KB NTFS allocation unit size. I now address the rest, add more tempdb data files, and move the tempdb log to the new Log drive. These are absolute minimums I should verify. Please check up on the aforementioned SQL Server VM best practices article for other recommendations and backgrounds.

Azure SQL Database

Now let me discuss Azure SQL Database as a platform-as-a-service or database-as-a-service offering. It is a managed database service, and you can purchase it in a single database or elastic pool model, which I talk about in more detail soon. There is no physical server instance available. There is a logical server instead. Thus features and

workloads that would reach out to the physical server level will not work. Similarly, while tempdb can be used via the workloads, there is no tempdb configuration available. Configuration comes with the service tier. There are no server configuration options or trace flags available either. It provides partial compatibility with full-fledged on-premise SQL Server environments. Due to this, it is advised to review what's supported and what's not, and generally it's a better choice for brand-new SQL Server solutions planned from scratch rather than forcing an on-premise migration to it. But of course, there are examples to this too. It also comes with on-premise Enterprise edition-only features, like transparent data encryption or read-only replica workload offloading, what otherwise always-on availability group clusters could provide on premise. While some automatic tuning options are available on premise in SQL 2017, like last-known good query plan enforcement, others are only available in Azure SQL Database, at least currently, for example, automatic index tuning. Of course, it is a fully tiered service, coming with different scale limits in regards to computing or storage capacity and performance. In order to be able to scale a service or resources, you must be able to monitor it properly to understand resource utilization and patterns. Azure SQL Database via the Azure portal provides built-in monitoring features, performance overview dashboards, performance recommendations, or advanced machine learning-based diagnostics like intelligent insights and automatic tuning. Query store that is otherwise available since SQL 2016 with on-premise deployments is available of course here too, and has since become a primary tool for workload optimization. Since it is a managed database offering, many diagnostic management views are replaced with Azure SQL Database-specific views that provide database-level data, for example, wait statistics at database instead of server-instance level implemented by the sys.dm_db_wait_stats view instead of the previously mentioned sys.dm_os_wait_stats view. Compared to the infrastructure-as-a-service model, platform-as-a-service database solution performance optimization essentially focuses on choosing the right service tier and everything that lives or scoped inside the database. That is database configuration, schema, indexing, and the workload itself. When troubleshooting performance problems, we don't need to bother with server configuration options, patching, or storage configurations. Those come with the service tier we chose. Thus understanding the service tiers and corresponding scale limits is crucial for performance optimization. I have mentioned service tiers many times. When purchasing a managed SQL Server service, you first need to choose a purchase model. Currently, there are two such models, DTU and vCore-based models. DTU was the first model, where DTU stands for database transaction units, which is a blend or mixture of CPU, I/O, and memory. One hundred DTU is twice as much as 50 DTU, and 200 DTU is twice as much as 100 DTU. Calculating with DTUs is relatively simple. You primarily need to compare them and evaluate what each compute size offers. These compute sizes are available within the service tiers, like Basic, Standard, and Premium. Each tier consists of multiple sizes, and each size provides varying scales of not just DTU, but number of sessions allowed, maximum database sizing, and storage capacity, also storage performance, and even underlying tempdb configuration and scale. The vCore model is quite recent and was designed to address the finetuning of CPU power and storage performance separately and in a more transparent way, since the DTU model, while simple, might be less transparent for resource optimization at different levels. There is an option to switch between these models, and for now, I use the DTU model in my examples. When

using the single database product type, the service tiers like Basic, Standard, and Premium determine the resource limits. Each tier then provides multiple compute sizes that you can choose from. For example, in the Standard tier, you can choose the S3 compute size, which currently comes with 100 maximum DTU, storage sizes ranging from 250 GB to 1 TB, and 2400 maximum concurrent sessions. Its underlying tempdb configuration, which you can't change and manage, comes with 1 tempdb data file with a fixed size of 32 GB. So the compute size determines the maximum DTU, storage, concurrent workers and sessions, along with tempdb configuration. If a chosen compute size proves to be not enough, you can either upscale to a higher compute size or even change to a higher service tier too, or further optimize the workload to fit into the chosen size. Azure tiers emphasize the importance of workload optimization, as unoptimized workloads will directly mean a higher cost for you. When planning a migration into Azure or planning a brand-new database solution natively in Azure, estimating the costs and choosing the purchase model along with the service tier can be a big challenge. In order to able to estimate all these, you must have some sort of performance baseline measurements already available on premise to calculate with. Unfortunately, this is not always available, and if there are hundreds of differently sized and mixed environments to be moved, it would be nice to have a set of tools that help in the estimations. There are at least two methods and tools that you can use for analyzing your current databases and workload characteristics to get recommendations about potential target Azure service tiers for those. The DTU calculator has been available for a while now. You need to collect perfmon traces with selected counters and then provide it as an input for the tool, which in return creates a report of recommended service tiers and compute sizes. The other more advanced one is part of the Data Migration Assistant, or DMA, tool, which is very useful in itself to analyze your current databases for a future migration to different targets like Azure SQL Database and find if there are any compatibility problems or breaking changes at schema or code level. There is a separate PowerShell script called SKU recommendation script included with DMA currently without the UI that you can use for more advanced and up-to-date recommendations, including the vCore model too. The script collects perfmon traces. The longer data collection gives more accurate results, and then that measurement is used as an input to provide up-to-date cost and sizing recommendations.

Elastic Pools

I talked about purchase models already, the DTU and the vCore model. There are two product types related to Azure SQL Database, single database and elastic pool. And now I'll discuss elastic pools in more detail. When using the DTU model, you can size each single database with separate compute sizes, like one database has an S2 size in the Standard tier, another database has a P1 size in the Premium tier, then you have multiple databases all having the S3 size in the Standard tier, and so on. If the assigned size proves to be inadequate, you need to upscale each database separately. When having many Azure SQL databases, there is an option to move them into elastic pools and choose a service tier and size for the assigned elastic pool instead. This way, resources are assigned to the entire pool and distributed among the databases dynamically. This also provides a much more flexible performance

management for databases. Elastic pools in the DTU model have so-called eDTUs, or elastic DTUs. These are similar to DTUs, but eDTUs are distributed and shared among the pooled databases. Also, a pool of a given size is more expensive than a single database, so in order to be cost effective, there are a couple of factors that need to be taken into account when planning elastic pools like what are the current average DTU usage across your databases and how the DTU peaks relate to the DTU averages. The elastic pools can be upscaled or downscaled in a similar way as single databases, and databases can be moved in between pools dynamically. Elastic pool sizing is similar to single database sizing. You choose service tiers and eDTU sizes that determine other scale limits for the entire pool like maximum storage per pool, maximum number of databases per pool, minimum and maximum eDTU for a database, and tempdb configuration and size. When having multiple databases to be considered for elastic pool rather than managing them individually, you can use the 1.5 multiplier in different ways for your calculations. One aspect is to calculate the sum of single database resources if the sum is 1.5 times more than the resources needed for an elastic pool, you can consider using a pool instead, as a pool's cost is about 1.5 times more than the same size for a single database. If you know the average DTU usage for the single databases, a question is how each database peaks compared to its average. If the peak is about 1.5 times more than its average, then again it can be an indication for using a pool instead. Speaking of peaks, an elastic pool is recommended for single databases that mostly do not have overlapping peaks. They peak at different times or at maximum two-thirds of the databases peak to their maximum resource limits at any given time. Remember, elastic pools are more expensive and resources are shared, so you need to count with performance while remaining cost effective. The fewer databases peak concurrently, the more cost effective the elastic pool option becomes. So if you know that overall your single databases have low average utilization and have occasional peaks which mostly do not overlap, this is a pattern that might indicate using an elastic pool instead. When choosing an elastic pool when sizes are identified, choose a size that otherwise provides the scales at other levels, too, including storage capacity, number of concurrent sessions, and so on. Now if you find that your peak times, DTU utilization patterns, and the number of single databases would indicate using an elastic pool instead, how to determine or at least estimate the correct size of the pool to be also cost effective? For this, first you need to evaluate the average and peak DTU utilizations correlated with the number of databases you have and take whichever is greater of the two measures. Next step is counting with the storage capacity for the pool and see which eDTU size provides that storage scale and take whichever is larger so far. And finally, since cost will be a factor for sure, use the Azure calculator to see which eDTU size provides the best value at this point. Managed services are tiered; therefore you always have an option to change tiers and sizes on the fly. Scaling up means that the higher tier is selected with more resources and higher scale limits. For example, you change compute size from S2 to S3 within the Standard tier, or move to Premium tier from Standard. Scaling down means that the resource utilization prove to be low enough that the higher service tier or size is not cost effective anymore, thus moving to a lower tier or size is possible. Scaling out is another option, just as on premise with the proper infrastructure and application design. Azure SQL Database provides an option to offload read-only, typically reporting workloads, to a replica, just as it is possible with always-on availability group clusters on premise, but only

in the Enterprise edition of SQL Server. A highly advanced option is database sharding to split data into multiple databases and manage them as different shards from the application. This has a separate toolset called Elastic Database Tools.

Demo: Estimating Service Tiers and Sizes

In this demo, I show you the tools used to estimate target Azure SQL database performance tiers and sizes, based on performance measurements in your on-premise environment, namely the DTU calculator and the SKU recommendation script, as part of the Data Migration Assistant tool. I also demonstrate how to work with performance tiers and sizes in creating Azure SQL Databases via the Azure portal, and when is it better to work with elastic pools instead of single databases? One of the tools available to try and estimate the target Azure SQL Database service tier and compute size in the DTU purchase model is the Azure SQL Database DTU calculator, which is an online tool. It essentially requires a perfmon trace with selected counters like processor time, logical disk counters, and database log bytes flushed per second as an input, and it generates a report with recommended sizes. In this example, I manually created a permon trace, which I then converted into a CSV format with the command line relog utility, specified the number of CPU cores for the tool, and uploaded the CSV file. In my example, the tool recommended the Premium P6 service tier and size, mainly due to IOPS requirements, and you can check estimations for CPU and log activity too. The other more advanced and up-to-date tool is shipped with the Data Migration Assistant, or DMA, tool. It is basically a PowerShell script that resides in the DMA install folder. It does not have a separate UI currently. Let's see how it works. I run the SkuRecommendationDataCollectionScript.ps1 script in a PowerShell window. I need to provide a couple of parameters like computer name and SQL server connection string, along with an output path where the trace file will be generated. This script collects performance measurement data in a specified time interval, which should be at least 2 minutes, but the longer the better to capture data of relevant workload activity and patterns. The generated trace file will then be provided as an input to the DmaCmd.exe utility, along with a couple of parameters, noting the source trace file location, where to put the result files, and whether I want an up-to-date Azure region-specific price information, or I am good with the offline mode for now. I run the utility for now, and also set that I want recommendations for the WideWorldImporters database in the source instance, but I could specify multiple databases too. I got a bunch of output files, HTML report files, along with TSV files, that is Tab Separated Values, that I can open in Excel. The file names indicate that I got recommendations both for Azure SQL Database and SQL managed instance. By opening the Azure SQL Database TSV file in Excel, I can see what options were evaluated for my selected WideWorldImporters database, what's recommended, and what were excluded by what rules. The same applies to the managed instance recommendations, which I show here for completeness. Opening the HTML report files shows more complete and a much nicer format to see what the recommended sizes are, and if I were to connect to Azure with the tool, I would not be able to see up-to-date pricing information, too. But I use the offline mode for now. I got a recommendation for Premium service tier with the P1 size

for my WideWorldImporters database based on that 2-minute performance data collection. Again, if you want more reliable and relevant sizing recommendations, collect data for a longer time period with valid workloads to have good data. The DMA tool can accept many parameters. Please review the official documentation about the use cases in more detail. Now let's create a couple of managed Azure SQL databases via the Azure portal to see what I can configure on them exactly regarding sizes and tiers. I first create an S0 database named database_1 that has 10 DTUs in the Standard tier. It costs around 12.65 Euros per month. I then create a couple more databases with the exact same sizing, S0 databases all having 10 DTUs in the Standard tier, so all together, I now have 15 such databases. By clicking in the database properties, I can see that these are single databases, no elastic pool assigned for them, and by default, the DTU utilization can be seen. Let's assume that these small databases have a low average DTU utilization per database. There will be peaks occasionally, and the peaks are at least 1.5 times higher than the averages. And what's also very important, the databases tend to peak at different times, so they mostly do not overlap with their peaks. As now I have at least 15 10-DTU databases and counting. Now I might be better off having an elastic pool assigned to them, an S1 with 100 eDTUs, if otherwise I am good from storage perspective. For now, I create an S1 100-eDTU elastic pool and add all my existing databases to it. The estimated combined price is now the same, or even a little bit smaller than what I would otherwise have with my single databases in total. Having less S0 10-DTU databases on the 15, and the elastic pool will not be too cost effective here. Similarly to single databases, since I now see that not much is going on with my databases in my S1 pool, I decided to downscale my elastic pool to a Basic tier one, and of course I then need to monitor performance. And this is the wrap up for our module. I explained key concepts in the Microsoft Azure cloud platform. What are the benefits of SQL Server solutions in a cloud platform like Microsoft Azure? I discussed the current SQL Server offerings in Azure, including SQL Server VMs, Azure SQL Database, and SQL managed instances. I discussed the main concept of tiered resources and the considerations for choosing a service tier and proper sizes. I detailed the best practices used when working with SQL Server virtual machines in Azure. I explained the key differences between SQL Server VMs and the Azure SQL Database and showed how to calculate with and estimate single database and elastic pool tiers and sizes.

Troubleshooting and Baselining the Environment

Troubleshooting and Baselining in SQL Server

Hello, and welcome back to my course, Managing SQL Server Database Performance. In this module, I show you tools that are available to troubleshoot performance problems in practice. I teach you how to use these tools and what are the common troubleshooting patterns that you can apply in your SQL Server environments. I also explain the importance of performance baselining so that you can identify new emerging problems and patterns more easily. When talking about troubleshooting, our aim is to understand, identify, and resolve problems with a selected set of tools specific to the technology. However, the problems can appear in different times and can have varying characteristics. The most straightforward scenario is when the problem is persistent, it occurs in an ongoing manner, or can be reproduced consistently at any time. This is the most fortunate scenario. We can throw an arsenal of tools at it and try different approaches as we wish, collect detailed data to see what's happening. It's more common though, especially with performance issues, that the problems appear in a random manner, or they are intermittent. It happened twice in the last six months then once last week, and we can't be sure when it appears again or if it reappears at all. We can't collect data in depth from every level continuously, as that might pose an overhead in itself in the production environment unnecessarily, so we need to be clever about when to collect what data for these intermittent problems. Another type of troubleshooting is the so-called root cause analysis, or RCA. A problem happened weeks or even months ago. No one had seen that before, and it has not been seen ever since. However, it caused an outage, or otherwise was serious enough to trigger an investigation into what happened exactly to be able to react to it next time or prevent it from happening again. The challenge is, of course, we might not have relevant data covering that event anymore. Monitoring data had not been persisted, or retention period has expired. In case you have a SQL Server-aware monitoring solution in place, you are in a better situation, but that solution still needs to be configured and customized accordingly, and you need the skills to understand the collected data and metrics. In the coming sections, I show you the toolset that we can apply in all of these scenarios, and SQL Server monitoring solutions work with these same tools and data as well. Talking about RCA, the retention period of persisted monitoring or log data is of high importance. The default settings of data retention, or the number of files to use before reusing them might not be adequate for investigations over a longer time period. You need to increase these, or in case of log files, persist the data manually by copying them over for archival purposes. Many times, a root cause analysis is not possible because relevant diagnostic or log data simply do not exist anymore. My module title also mentions baselining. What's that's exactly? Previously, I discussed data collection during time periods when problems occur. However, if you collect data outside of problematic time periods when things are okay, then you

have a pattern of performance metrics that you can consider normal. There might be differences in what you call normal operation, depending on the particular environment and workloads, so essentially the baselines can differ in each environment. Another way of collecting baselines is when focusing on selected time periods only, when you expect outstanding workloads to see how that scales later on, when data volume or number of users increases, for example. If you're familiar with existing normal patterns, then even a slight change in patterns might be very useful to watch out for, way before things become serious enough to trigger system-wide alerts. I discuss the following diagnostic tools and options available in SQL Server that help you both in performance troubleshooting and baselining, whether it's on premise or infrastructure as a service or Azure SQL Database. These are wait statistics, which is the basis of performance troubleshooting and baselining in SQL Server. Many types of data collections rely on dynamic management views, or DMVs, which are in-memory views of system behavior or performance metrics, but they are lost with a service restart, so persisting the results can be very helpful as part of your monitoring. The query store, which is available in SQL 2016 and above, and enables you to track query patterns over a period of time, look back at query runtime statistics along with query plans, and in SQL 2017 with query wait statistics too. The system_health session trace available since SQL 2012 is a black box-type flight recorder which runs out of the box and collects many system-wise diagnostic data, data of various error conditions and deadlock graphs too. Performance Monitor, or perfmon, of course, where the performance counters, if selected carefully, can provide a full context and details of performance behaviors over a period of time.

Wait Statistics in Practice

In this section, I discuss wait statistics in practice. Looking back at my timeline presentation of the types of performance problems, wait statistics can be applied in all those scenarios, whether it's a persistent, intermittent, or an ad hoc problem requiring root cause analysis. The closer you are to the problem occurrence, the better your chance for capturing relevant wait statistics data that can be correlated directly with the problem. When checking higher-level wait statistics for intermittent or root cause analysis, it becomes more or less a health check approach, which might give away red flags, in hind sight. But having persisted diagnostic data covering those events would be much better. Now let's see what are the SQL Server tools and methods for capturing wait statistics. Sys.dm_os_wait_stats view, the classic one, useful with on-premise or infrastructure- as-a-service full instances and provides server instance level data. Sys.dm_db -wait_stats view, this is a newer one, and is applicable to Azure SQL Database, where there is no physical instance available and you want to get wait stats for your database context instead. Sys.dm_exec_session_wait_stats, a newer one since SQL 2016, and provides session-level wait statistics. Can be useful when troubleshooting in an environment where of course many sessions run in parallel, you have the session ID, and want to narrow down waits to that session ID, or you try something out in SQL Management Studio, run a batch, and get wait statistics for you own test session only. Previously it was only possible with custom extended event traces. The Query Store in SQL 2017. The Query Store is enhanced with tracking query-level wait

statistics. And as this is a persisting store, it enables you to do all types of troubleshooting, whether it's persistent, intermittent, or RCA. Of course, with custom traces, extended event traces, or perfmon, for example, you can capture wait statistics data too. Wait statistics can be collected from different levels, or scopes, within SQL Server. The sys.dm_os_wait_stats view provides server instance-level cumulative statistics, which is useful with on-premise full physical instances, but it does not tell you who contributed to those waits and when. The sys.dm_db_wait_stats view provides database-level statistics, but only in Azure SQL Database when there is no physical instance available. Instead each database is provided as a database-as-a-service offering. Session-level wait statistics can be collected with the sys.dm_exec_session_wait_stats view in SQL 2016 and above, or with custom traces in all versions. Query level wait statistics can be collected with the SQL 2017 version of the query store or with custom traces in all versions. This also gives you a layered approach, and I recommend a top-down troubleshooting method as usual, first checking wait stats at server level and then digging down into lower layers if necessary. Now let's see why server-level wait statistics is very useful in how it can help you in performance troubleshooting exactly. The sys.dm_os_wait_stats view provides cumulative statistics for all wait types, and there are hundreds of them since the last restart of the server. So next time the service is restarted, this data is gone. It can be cleared manually though at any time, which can be useful when troubleshooting an emerging problem after a long period of silence. So the server has been running okay for months, and then suddenly problems start to occur in the morning. Then clearing the wait stats can provide more accurate results at that point. Also, as this is a cumulative statistic, providing metrics like number of tasks for a wait type or wait count, wait time, maximum wait time, and signal wait time in milliseconds, it does not tell you who, which workloads contributed to wait type, and when. If the server has been running for months, a particular wait type might have emerged shortly after service startup, but we don't know if that wait type is still relevant or not right now. Also, it would be nice to have more metrics like average wait times, resource wait times, and also browsing through hundreds of wait types is not fun, after all. Querying my SQL 2017 instance results in 928 entries currently. That is, that many different wait types exist. How do I know what each indicates and means exactly? So what is good for then? It's a primary tool for server health checks. Even if just collecting it is a snapshot, it gives an overall picture of what the server waits for in general. It is also useful for collecting baselines, when persisting the results for tracking changes and trend analysis, and of course it can directly identify problems or troubleshooting entry points. While this view is raw data, the SQL Server community uses a custom query pattern that works out of this view but provides a much more meaningful output that I show you next and in demos later in this module. This custom query is a must have, and should be a part of your SQL Server troubleshooting toolset every time. It is provided by sqlskills.com and namely by Paul Randal. Please visit their site and the Bitly link above for direct references on how this works and also on wait statistics analysis and troubleshooting deep dives. You can find highly useful and detailed resources in there. So what this query does that enhances the raw wait statistics data? It shows the prevalent waits only, the top 95% waits by default, which you are free to change, of course. It calculates average wait times, resource wait times provided in seconds instead of milliseconds, it excludes tons of good, or benign, wait types that otherwise would skew the statistics for our troubleshooting needs. I always run this as a

snapshot data collection or schedule it with agent jobs, persist its output into a table for trend analysis later on, where you can query the differences in wait statistics over a period of time. It's an easy and built-in way to start performance troubleshooting in all versions of SQL Server. With a little modification of using sys.dm_db_wait_stats instead of sys.dm_os_wait_stats, you can use this same query pattern in Azure SQL Database too.

IO and CPU Related Wait Type Patterns

The aim with our custom query is to have a list of prevalent wait types that tell us what SQL Server waits for exactly. Now let me show you a couple of notable wait types that you can encounter in production environments, along with a short explanation of what they mean and what problems they indicate. Of course, there are many more such wait types. Please check out the wait type libraries for further details that I reference later on in this module, and of course check up on your very own environment. PAGIOLATCH type waits and WRITELOG. These are I/O waits, which mean that SQL Server waits for an I/O resource to complete. PAGIOLATCH waits appear as SH, or shared, and EX, or exclusive, mode waits. These occur when reading in a page from physical disk into buffer pool memory, and then in the pages to be read or modified, you see the shared or exclusive mode with these waits. When this wait type becomes prevalent with high wait counts or bad average resource wait times, it means that there are lots of I/O operations happening with potentially subpar or plain bad I/O performance. But why is that exactly can be rooted back to multiple things? By far, the most frequent reason is bad SQL Server memory configuration or memory pressure problems in the environment, causing an increased I/O load and pressure on the storage subsystem. So the bad I/O performance is many times the symptom of memory issues. You need to check up on the configured buffer pool memory size, if it is sized properly to your workload, if server configuration settings like max and min server memory settings, maybe lock pages in memory are configured properly, and if there are signs of memory pressure. Many time, the increased I/O load, the increased memory usage is rooted back to bad index usage, missing indexes, or out-of-date statistics. Of course, if all these are okay, the storage subsystem can have performance problems natively too. You need to then check up on the storage performance with further traces and involve the storage team for further investigation. Or validate the storage tier and type you use in Azure. WRITELOG is another common wait type. It is an I/O wait when writing the transaction log file. If this is prevalent, especially with bad resource wait averages, you need to check up on whether the transaction log file of your database resides in a good performing disk. Log files are separated from other database files, and generally apply all the storage best practices that optimize sequential write I/O performance for transaction log writes under SQL Server. In Azure infrastructure as a service, it can mean hosting the transaction log file on a separate and properly tiered SSD managed disk. There is another less common indication when having lots of small transactions causing an internal limitation again, resulting in this wait type. Having bad write log wait directly impact your transaction performance in SQL Server, you need to deal with this for sure. Talked about out I/O. Now let's see a few CPU-related wait types. CXPACKET and SOS_SCHEDULER_YIELD. When you see these wait types, it does not mean that you have CPU

utilization problems or high CPU problems. Still you can have workload performance issues. CXPACKET means that there is parallelism happening on the server, which can be a good thing if expected, or a bad thing if it is excessive. When having this with a high wait count or bad averages, first thing to know if this is expected or where it comes from. When having reporting workloads doing large scans or sorts, this can be expected. When having index usage problems, and these costly operations are done under the covers in parallel, you need to address the workload and the indexing to support the workloads. As an addition, later patch versions of SQL Server introduced the related wait type of CXCONSUMER, which then become the good parallelism wait separately. The custom script now filters them out as benign waits, so essentially what you are left with the CXPACKET waits might need attention. When seeing CXPACKET, I always check up on server configuration settings like max degree of parallelism and cost threshold for parallelism if these are set according to best practices or otherwise changed for some reason. But do not turn parallelism off as a first reaction. In SQL 2016 and above, I also check up on database scoped configuration for MAXDOP. When suspecting index usage problems but can't change indexing, I also check up on index maintenance jobs, especially statistics updates, if those are set up and run properly to address skewed parallelism problems. SOS_SCHEDULER_YIELD indicates persistent CPU utilization. The threads use up the 4 ms quantum on the scheduler, then yield it, and they wait for no resources. There are no resource waits. It can be an indication of large scans again, but in a virtualized environment when this is prevalent, it can indicate host server CPU configuration problems too, or overloaded host. In any case, I would recommend checking up on server configuration settings, like an example I'll show you soon, and also tracking this at workload level might be necessary to draw further conclusions.

OS, Locking, and Data Page Related Wait Type Patterns

When having preemptive wait types appearing, it means that the waits occur in the underlying OS that in Windows, in our case. Remember, SQL Server uses a non-preemptive threading model. Windows uses the preemptive model, so whenever a call waits in Windows, SQL Server knows this as a preemptive wait. For example, PREEMPTIVE_OS_WRITEFILEGATHER indicates problems around file autogrowth, as it generally occurs when doing zero file initialization and NTFS level, and SQL Server calls the appropriate API functions. When you see this, especially with bad average resource wait times, you can suspect that instant file initialization is not turned on, or otherwise autogrowth settings are not optimal and cause an overhead. So check up on whether instant file initialization is on, if database autogrowth settings are optimal, look for large growth values or percentage growth settings, and again, the storage subsystem can have problems too, anything that impacts autogrowth performance outside of SQL Server. Another interesting wait type is PREEMPTIVE_OS_AUTHENTICATIONOPS. This can pop up, but rarely gets to the top. I saw this once as becoming highly prevalent, and it indicated an authentication problem against domain controller servers, so when this is causing problems, you will most probably need to involve the domain or network team for further investigation. Lock-type waits, whether in shared lock mode or exclusive

mode, generally imply workload and/or index usage problems. Workload problems can also include using a more restrictive transition isolation level, for example. Accumulating these waits needs to be investigated, as this also means transaction performance problems and customer complaints for sure. We can also encounter deadlocks when having all sorts of blocking and locking problems. All these require workload-level traces and investigation. Using the Query Store can help here a lot to check up on query plans and runtime statistics or checking up on deadlocks in the system_health session trace too. Modifying and optimizing your query workload might be needed as well. When having system-level problems, memory, tempdb, and I/O bottlenecks, holding onto locks longer can also occur, and the otherwise low amount of locking problems can increase significantly, so doing a top-down troubleshooting approach is beneficial here too. PAGELATCH-type waits are often mistaken for I/O waits like PAGEIOLATCH, but these do not have anything to do with I/O. These are in-memory page access-related waits. These often occur when having lots of page splits, insert hotspot problems, or when the wait resources point to tempdb can indicate the well-known tempdb contention problems. Tracing these at workload and indexing level might be needed to draw further conclusions.

Wait Statistics Patterns in Troubleshooting

Let me show you a real-life example of a custom solution when using server-level wait statistics, also when baselining with wait statistics can be helpful. The wait types are collected with our custom script, and the order means prevalence. The higher the wait type is in the list, the more prevalent it is. Before and after shows the top 95% prevalent wait types for the very same application, the very same database. Before column shows the baseline we have. After column reflects some changes that the customer did, but we do not know what they did exactly. We are after the fact. What we know is that after also means major performance problems and end user complaints of all sorts. It turned out that the environment has been migrated. Before is a physical environment, lots of CPU cores and memory, but shared environment. After is a downside virtual machine, but dedicated to the application. First thing to notice is CXPACKET, which was the most prevalent wait type with high wait counts and low average wait times, is now almost gone. It is now way less prevalent with a fraction of earlier wait counts, but now with very bad averages. CXCONSUMER waits are not filtered out in these collections; they do not appear either. We don't know what happened exactly, but something happened to parallelism on this server for sure. It's almost like parallelism is not present anymore, which is a bad thing at first, as our solution expects and benefits from parallelism due to its mixed workloads primarily. Another notable thing is existing but low occurrences of locking problems, also coming with deadlocks, which are rooted back to some workload and indexing issues, are not much more prevalent with really bad average wait times. On top of that, new wait types emerged or increased to prevalence, SOS_SCHEDULER_YIELD and PAGEIOLATCH_SH. We have multiple problems here. Parallelism is less prevalent. As it turned out server level cost threshold for parallelism has been increased to a very high range from the default five, essentially meaning that parallelism does not kick in for the majority of our workloads. It's almost like switching

parallelism off. However, this might have been done on purpose as a quick reaction to having parallelism-related problems already. We don't have information on this unfortunately. We don't have historical data. This can indicate the appearance or increase of SOS_SCHEDULER_YIELD waits, but it turned out this is a VM, so checking up on host-level performance might be needed too later on. The increased lock problems might be due to holding onto locks much longer, having slower serial plans, but where these are coming from in the first place, really do require indexing optimizations as we know it or applying missing indexes too. So the fact of running in this new downscaled server triggers the need of the sooner than later optimization of the known workload problems. Also tempdb's having very bad write I/O latency values, it can contribute to these increased lock waits too. PAGEIOLATCH types also indicate that the downscale might also have introduced periodic buffer pool memory pressure. This needs to be traced in detail over a longer period of time, as it is not visible right at the moment. So the areas to be addressed are at first glimpse server configuration for parallelism. Cost threshold for parallelism could be lowered a bit and see impact, plus MAXDOP must be adjusted too, accordingly. Missing indexes behind the already- identified workload problems to reduce the need of unnecessary scans. Making sure that index statistics updates are scheduled and run properly. VM and VM host configuration and performance tracing system performance. Validating if the new VM and SQL Server buffer pool memory size is valid. At the end it might turn out that the larger VM size must also be used. A perfmon trace should be collected to see resource utilization trends, both at system and SQL Server levels over a longer period of time. Then if possible, using the Query Store would help in tracking down query plans with runtime statistics, along with query wait statistics to help in identifying the source workloads. When doing wait statistics analysis, always focus on the bad waits, as waits are normal due to the SQL Server threading model. Focus on the prevalent waits, check wait counts, and average wait times. On practical terms, out of the hundreds of available wait types, only a dozen or two will pop up as prevalent wait types in production environments. Look for changes in patterns. If you are used to the patterns of your existing application, a slight change in that will come as a red flag. There are wait types that you cannot really live with. That is, you need to address those first, like THREADPOOL and RESOURCE_SEMAPHORE waits, for example. These are red flags, or poison waits. How to know which wait type means what and how to troubleshoot each? There are comprehensive wait type libraries in the official Microsoft documentation of the sys.dm_os_wait_stats view, or at sqlskill.com. So whenever you see a prevalent wait type, go check what it means in detail at the above links. Don't focus on memorizing these. At first, just get production samples with a custom script, see how your environments look like as wait type patterns, and see how they change over time or by changing the environment on upgrades or other application or data-related modifications. This really requires doing it to see how it works, and get as many samples as you can. Get familiar with it.

Demo: Wait Statistics Analysis with Custom Query

In this demo, I'll show you how to collect and analyze server-level wait statistics in an on-premise SQL Server environment. I also show you how to collect and analyze database-level wait statistics in an Azure SQL Database

environment. In my demo, I will showcase how to use the custom wait statistics script for both scenarios. I am now connected to an Azure SQL Database on pluraldbserver1, and this is a version of the Wide World Importers sample database. Let's see our custom server instance wait statistics query using the sys.dm_os_wait_stats view. It calculates resource wait time by subtracting signal wait time from wait time and converts wait times to seconds. Also calculates average wait times and percentages. There are tons of exclusions, the benign wait types like WAITFOR. If you used WAITFOR in your workload and we didn't filter it out, it would be too prevalent and skew our results. Let's run this. While this works, it gives me strange results, not the usual wait types I expect. So this DMV works here, too, but we should use the sys.dm_db_wait_stats view to get the desired wait statistics for our Azure SQL Database context instead. Let's run the query again, but with the sys.dm_db_wait_stats view this time. I get my desired results. I ran a workload prior to this in this Basic-tiered Wide World Importers database doing large scans, and that's where SCHEDULER_YIELD is indeed on the top with bad average wait time of 91 ms. I have some bad I/O completion wait type too with an average wait time of 150 ms. That can be caused by transaction log reads. It is not the data page I/O wait. Also RESOURCE_SEMAPHORE indicating memory allocation problems, potentially, so using a Basic-tiered database was a good idea for experimenting a bit, but I might not use it in production with my workload. I would try something in a higher tier and compare the wait statistics. I now run this query in a full on-premise or infrastructure-as-a-service instance using the sys.dm_os_wait_stats view, and prior to this, I also ran my workload. This gives me different patterns, seemingly. I/O completion is here too, but with much less average wait time. It's about 7 ms. I have preemptive waits suspiciously doing some Windows-level file operations, and I think this is due to the file string feature here. And there is no file string feature in Azure SQL Database, if I make a comparison. PAGEIOLATCH was probably due to the initial reading of the pages from disk, but the average of 4.5 ms is okay, so I'm not complaining yet. As I move on with testing my workload, I need to capture wait stats again and again and see how it changes over time and how the trend looks like.

IO Performance Troubleshooting with DMVs

In this section, I discuss SQL Server dynamic management views, or DMVs. Actually, I have been using DMVs within this course throughout, whether it's sys.dom_os_wait_stats or sys.dm_os_waiting_tasks, these version-specific management views provide all sorts of insights into virtually all areas within SQL Server. Therefore, you can use these for troubleshooting of all sorts of problems. However, many contain either cumulative or current data, so if you want to see the changes and trends over time that might help you in tracking down problems in the past, you need to persist the results into a database for later analysis. So DMVs provide views of internal data in SQL Server, but they are in-memory views. When restarting the server, their content is gone. There are hundreds of different DMVs for different areas or components within SQL Server. I discuss SQL OS-related DMVs like sys.dm_os_wait_stats within this course quite frequently, but there are DMVs specific for I/O management, databases, indexes, the server instance, transactions, and so on. Another favorite DMV of ours when troubleshooting performance problems

exposes database file-level I/O statistics. It is named sys.dm_io_virtual_file_stats. Similar to the well-known wait statistics DMV, it contains data that we could parse further to be more readable, thus the SQL Server community tends to use a custom script instead that works out of this view, and this is again provided by sqlskills.com and Paul Randal. Please check the above URL and Bitly link. This custom script queries this DMV and calculates file-level I/O latency metrics and average read and write bytes. This is extremely useful when checking up on a server to see which database files on which drives might have subpar I/O performance without complex traces to start with. This can be a little bit misleading though. As this is a DMV, it contains averages since the last restart of the server, potentially over a long period of time. Therefore, if you see more or less normal I/O latency values here, the server still can have bad I/O latency spikes randomly. When you see bad values here, those need to be investigated with the perfmon trace, for example, to see how the trend looks like. This custom script is very useful, and is one of the easiest and quickest methods to find, for example, tempdb I/O and configuration problems.

Demo: IO Performance Analysis with Custom Query

In this demo, I'll show you a custom script to check up on SQL Server database file-level I/O statistics. I now show you how to run another custom query to check SQL Server file-level I/O statistics using the sys.dm_io_virtual_file_stats view. I used it against my Basic-tiered Wide World Importers database after I ran my workload and collected the database-level wait statistics that showed subpar I/O waits, among others. The query calculates latency metrics in milliseconds, and I am using the DB_ID function in this query. I get results for the two database files of my database, the data and the log file, and all the latency values are in a bad range, hundreds of milliseconds, so it's not that good so far. I now run this against my full server instance, but I changed my script to get I/O statistics for all database files on the instance. Thus I use the sys.master_files view. There is no such view in Azure SQL Database. I get the results, and with the same workload but with a totally different setup and resources. The latencies here look way better. The latencies are a few milliseconds overall. Actually, so far the file-level I/O average latencies nicely correlate with the I/O wait type averages in both the server instance and the Azure SQL Database scenario.

Query Store

In this section, I discuss the SQL Server Query Store. The SQL Server Query Store is a powerful feature to be used in all troubleshooting scenarios. Besides those, it can be used for a variety of other use cases, and consider this more like a developer tool rather than a DBA tool. Actually, it should be used by developers more often to get direct feedback on their workload performance and characteristics from production environments. That way, they could see the impact of the queries more easily. I noted that the Query Store is a SQL 2016 feature and has been enhanced with wait statistics data in SQL 2017. However, with on-premise or infrastructure- as-a-service environments, it's

turned off by default. You need to turn it on manually on each database. In Azure SQL Database, it's turned on by default. The Query Store is a database-level persistent store. The collected data is stored in the user database where it is turned on. The Query Store contains query statements, query plans, and their runtime statistics, including number of rows accessed, CPU and memory usage, I/O statistics, and so on. SQL 2017 added query-level wait statistics as an enhancement. While it is extremely useful, you can get valuable reports and statistics out of the Query Store to have development and query optimization. You can report the most resource-intensive workloads, whether certain workload patterns were successful, failed with exception, or maybe timed out within specified time intervals. In hindsight therefore, this is a highly useful tool for troubleshooting intermittent performance problems and RCAs. You can track how the workload patterns and the query plans changed over time, maybe due to upgrades and changes in the application. Query plan regressions when query plans change for the worse, for example, when migrating a database to a newer platform version or when changing database compatibility level, can be tracked with the Query Store. Query Store and the insights it provides should be used by the developers more often, as it gives direct comparable information on how the workloads perform in the actual environment over time.

Demo: Using the Query Store for Query Performance Analysis

In this demo, I'll show you how to work with the Query Store in Azure SQL Database to track query plans and query performance. I am connected to my Azure SQL Wide World Importers database where the Query Store feature is turned on by default. I am interested in the Website.SearchForPeople stored procedure runtime statistics over a period of time, in this case on a given day. I am using multiple Query Store views to join to get query statements, plans, and runtime statistics. As where criteria, I add the time period and the OBJECT_ID of the SearchForPeople stored procedure. I get the results that show on the given day in which time Intervals, what execution patterns were used with SearchForPeople. Like between 8:00 and 9:00 am it was executed more than 200 times, the particular plan and query within the stored procedure. It shows me the runtime stats of CPU usage and also if the particular plan was serial or parallel. And I see the dop, or degree of parallelism values are all 1, meaning it was a serial plan. Clicking on the plan XML, I can research the execution plan that was run at the time. I now run this same query, but I also join in the new SQL 2017 sys.query_store_wait_stats view to get query wait statistics with my SearchForPeople stored procedure in the very same time period. The results show that we have wait categories instead of single wait types like in the server or database-level wait statistics. So we have buffer, I/O, or CPU waits, which can consist of multiple wait types, actually. I now see that some of my executions were aborted, and indeed I can sort some of the executions of this stored procedure. To get an idea how these query-level wait categories look like in detail, please check the official documentation of the sys.query_store_wait_stats view. To get more insights from the Query Store, you can use, for example, Power BI, either directly or by exporting the query result in to CSV format and import it into a Power BI dataset offline. For example, you can display the number of executions and selected runtime statistics of our SearchForPeople stored procedure over time. You can filter for execution type, whether it was successful, failed

with exception, or aborted. Also, you can filter for time ranges. The columns show the number of executions, and the lines show selected runtime metrics like maximum of row count. The below bubble chart shows what was the execution distribution of the stored procedure by the hours within a day, and you can click on a selected date to get filtered results. The bubble sizes correlate with durations, so the larger the bubble, the longer it took to execute the stored procedure. You can also play around with colors, if you like. It can also be useful to catch bugs. Seemingly, the maximum of row count has a high value for a long time. Then it drops to close to zero value. Indeed, at that point a fix was applied because the stored procedure had a bug to scan a large number of records unnecessarily from another table. And that line of code was removed and a new version was released on that day.

System_health Session Trace

In this section I talk about the system_health session trace. The system_health session trace is a built-in extended event trace available in SQL Server since SQL 2012, and acts as a flight recorder of erroneous activity for server-level events. You can change the definition of this trace session to increase the number of trace files, for example, to have a higher data retention. It is useful for all types of troubleshooting. I mainly use it for root cause analysis purposes; however, when troubleshooting locking and blocking problems that also come with deadlocks, it is very convenient to find the deadlock graphs in the trace without any extra configuration to provide them to the developers. The system_health session trace is an extended event trace running automatically in your SQL Server instance. The trace files reside in the SQL log folder by default. The file names start with System_health and the extension is XEL. By default, there are four such trace files, and they are being overwritten in a circular manner, or rolled over. Thus to increase the data retention, you can choose to increase the number of trace files or the maximum size of each file, or you can choose to copy off the files periodically for archival purposes. The traces contain all sorts of system- level diagnostic data and events, including selected errors occurring at server level, CPU and memory-related error conditions, the output of the sp_server_diagnostics stored procedure that was introduced in SQL 2012 and can be run manually too at any time. It creates XML report outputs of system resource utilization and overall system_health statuses. And deadlock graphs. It's very useful for RCA purposes if otherwise data covers the problematic time period and the traces can be analyzed in SSMS directly, but there are custom solutions parsing these traces with custom reporting services reports too. Getting deadlock graphs out of it directly can be extremely handy, as otherwise you would need to set trace flags or run custom traces to capture them.

Demo: Using the System_health Session Trace for Troubleshooting

In this demo, I'll show you the system_health session trace definition and use SSMS to analyze the traces. I am connected to my full server instance, and let's first see a stored procedure called sp_server_diagnostics. I run it directly from SSMS, and it generates system_health statuses for various areas like query processing, resource

utilization, I/O subsystem, and provides an overall health state. Here it is clean. It can be run at any time, and it gives a snapshot of overall system health. You can explore each category by checking the XML results. I now check what's in the system_health session trace located in SSMS, and click on the system_health session and click Properties. It collects selected events, range of errors, security and connectivity errors, SP server diagnostics results, wait info, and XML deadlock graphs, among others. The trace collects data into trace files and the memory ring buffer. In SSMS, I can select the trace file target, choose View Target Data, and analyze the traces on the fly. I now filter for sp_server_diagnostics results where the state is not equal to CLEAN, and I have one occurrence of such event so far. Let me find deadlocks in the trace, as users complain about query timeout problems too, or developers do not log deadlock events properly in the application logs. I have one deadlock report in the trace so far, which I generated deliberately, for that matter, and I can see or export the XML definition for further analysis. Another way to see the very same trace data is by using grouping, and then I can check the number of trace entries per category. For example, I now also track the restored database event with wait info.

Perfmon Traces

In this section, I discuss Windows Performance Monitor traces. Perfmon traces are best used when tracing ongoing or persistent problems. However, you can use perfmon indirectly for intermittent and RCA problems too. When having intermittent problems but you identify the problem pattern, like it tends to occur Mondays, then running a perfmon cover in that time period might cover the emerging issue too, if you're lucky. For RCAs, when persistent perfmon data collections are available, you can simply look back at the data, or if those are not available, doing baselines with perfmon might point out tendencies that otherwise you have not noticed until then, and they might have contributed to the RCA problem in the first place. You need to run a perfmon trace to monitor performance over a period of time and to see metrics in context. In a SQL Server on-premise or infrastructure-as-a-service environment, both system and SQL Server-specific objects and counters need to be added to the trace, so when tracing high CPU utilization problems at system level, for example, the SQL Server workload patterns are also correlated, like the rate of full scans, temp table activity, the rate of query plan compilations, or the number of blocked processes with system object counters like processor memory and LogicalDisk. Perfmon traces are mostly useful when tracing the problems as they happen or for baselining purposes. You can configure a perfmon trace by setting it up in the Windows Performance Monitor tool in Windows, and at the desired counters manually, or work from a template. Perfmon traces can also be collected from a remote computer having the necessary permissions. You can also use PowerShell scripts to set up perfmon data collections. For all of this, you don't need to connect to the SQL Server instance. The trace is set up at OS level. However, SQL Server also exposed cached counters that are SQL Server counters, and you can query the counter values at any time in the sys.dm_os_performance_counters view when connected to the instance. So to check a single SQL Server metric, depending on the metric type, you can simply query the metric value directly from this view, for example, page life expectancy. Of course, this way, you have

a metric value at the given point in time only as a snapshot value. When tracing a SQL Server instance with perfmon, there is a set of objects and counters that is recommended to be added, both at OS and SQL Server level, at the minimum. The OS counters definitely should contain processor, memory, and LogicalDisk counters, at least. Processor contains CPUs-related metrics for CPU utilization. Memory contains metrics to see how much memory is available at system level, among others. LogicalDisk contains the important disk I/O metrics for each drive, including disk I/O latency measures, both for reads and writes, also I/O throughput and metrics to see how busy a disk is. Disk I/O latency values, average disk sec/read, and average disk sec/write are measures to see if you have I/O performance problems on the SQL Server drives hosting data, transaction log, and tempdb files. Bad latency values will also pop up as PAGEIOLATCH or WRITELOGWAITS in wait statistics. SQL Server has many counters exposed, including memory management, data access patterns, database file management, or counters telling you how busy the SQL Server instance is. Analyzing perfmon traces can be tedious manually in the Windows Performance Monitor UI, but this is sometimes required to see the lower-level details to focus on a shorter period of time with selected counters only to correlate with each other. When you need quick analysis to see the big picture and overall trends, you can use some tools to parse the trace and provide the report output. Such very useful tool is the PAL utility available at the URL above. You provide the perfmon trace file as an input, can even select technology-specific templates with built-in thresholds and explanations, and the tool generates an HTML report with counter-specific charts, detailed explanations, along with warnings and timestamps.

Interesting SQL Server Counters

Let me highlight a few interesting perfmon counters out of the hundreds available that you can use. By far one of the most widely used counters is the Page Life Expectancy, or PLE, counter within the SQL Server buffer manager object. When setting up a perfmon trace, the entire buffer manager object can be added to the trace. It is measured in seconds and tells you how long a cached data page resides in the buffer pool memory on average. The higher this value, the better. Low PLE values indicate buffer pool memory pressure problems that can result in I/O performance problems too. Processes blocked within the SQL Server General Statistics object tells you, as the name suggests, the number of blocked processes currently, which can be useful when troubleshooting intermittent blocking problems as a trend. Batch Requests per second within the SQL statistics object indicates overall how busy your SQL Server is. The higher this value, the busier your server is, and as a result, you can see increasing CPU utilization correlated with this metric. SQL attention rate with the SQL Statistics object indicates requests terminated by the client for some reason. It can be interesting to correlate this with end users reporting timeouts or canceling long-running requests. Full scans per second within the Access Methods object can be interesting to correlate with parallelism, CPU utilization, and blocking problems over a period of time, to see the rate of full table scans that might cause such problems, maybe due to index usage problems. Total server memory within the Memory Manager object shows the

total committed memory by the SQL Server process. When setting up your perfmon trace, the entire Memory Manager object can be added to the trace, actually.

Page Life Expectancy Patterns

Talking about PLE being an important metric to see if SQL Server buffer pool memory is stable or have memory pressure problems, I noted that the aim is keeping this value in high ranges. High and even increasing PLE values, again measured in seconds, are good. Constant low values or sharply dropping values to low ranges are bad. Let me show you examples of notable PLE patterns that you can see in perfmon traces. The first one indicates PLE values in very high ranges, around 190, 000 seconds and increasing. This means that the buffer pool memory is stable and is sized properly to your workload needs. There is no sign of memory pressure. When checking the server wait statistics, and let's assume this is a dedicated SQL Server environment, you most probably won't see PAGEIOLATCH waits, and other perfmon disk I/O-related metrics like average disk sec/read and average disk sec/write will be okay as well, most probably. In the second case, PLE is in a constant low range, in the few-hundred seconds range, below 1000 seconds. Where a red line indicates a threshold I set for alerting when PLE drops below 1000 here, alerts will be generated. Assume that our instance has 64 GB buffer pool memory configured and uses all of it while having this PLE pattern. In about every 500 seconds on average, that is in 8 minutes, the entire 64 GB of buffer pool memory is flushed to disk, as pages constantly need to be read in from disk and written out to disk. This is not so good, and as a result, you most probably will see increased PAGEIOLATCH waits and subpar I/O latency values, along with overall bad performance in your instance. The third pattern is when PLE tends to be in higher ranges but periodically drops into very low ranges and stays there for longer time periods indicating potential outstanding workloads running, for example, with a financial application during monthly closing periods. These drops should be investigated if the outstanding workload can be optimized at code level or with indexes, or whether you can increase the buffer pool memory size to better tolerate these outstanding time periods.

Demo: Using and Analyzing Perfmon Traces

In this demo, I create a perfmon trace manually with Windows Performance Monitor and show you the counters to be added for a generic SQL Server performance trace. I then use the PAL utility to parse the trace and generate a report out of it. I then show you how to use the SQL Server cached counters within the sys.dm_os_performance_counters view to check up on metrics directly when connected to your SQL Server instance. As an extra, I use the relog.exe command line utility to export the perfmon trace into a SQL Server Database for custom reporting. Let me first show you the cached counters view sys.dm_os_performance_counters. When I query from it, I get all SQL Server- related performance counters as a result. Let me filter for page life expectancy that I am interested in, and as a snapshot value, it is 2364 seconds. Let me query it again and again, and I can see the value increases slowly. I now collect the

full perfmon trace with Windows Performance Monitor. I add all necessary system and SQL Server counters to the trace, leave the time interval at 15 seconds. When tracing for long time periods like days, you can increase this to a higher value like 30 seconds. Trace collection now starts. I now use the PAL utility to parse the trace file, set sources, temp parameters, and threshold file to be used, and run the tool to generate an HTML report. I now have a nice HTML report with detailed explanations for all counters, thresholds, warnings, and charts that you can easily copy off for even an executive summary. While this trace does not show too many problems, let me have another example as a real-life production sample. This is an example for having constantly bad PLE values in extremely low ranges, and as a result, with bad I/O performance. If you want more flexibility with perfmon trace analysis in a custom manner, you can choose to create your own reporting and analysis solution. There is a nice little command line utility called relog.exe that can be used to convert perfmon traces in between different formats. If you set up an ODBC data source name or DSN at system level pointing to a SQL Server custom database, you can also use relog to export all or selected counters from the source perfmon trace to a database where you can analyze the trace with SQL queries directly and maybe build your own reporting services reports around it.

Module Summary

And that's the wrap up for this module. SQL Server performance troubleshooting is quite an extensive and complex topic, and you need to practice it as much as you can to see how things really work and get familiar with real-life SQL Server performance patterns. I discussed the various types of performance problems and the toolset that you should use to address those problems. I talked about wait statistics in practice and showed how to use a nice custom query for starting your performance investigations. I talked about using another custom DMV query that you can use to identify SQL Server I/O related problems. I explained why the Query Store should be turned on in your production environments and how it can help you as a developer or other developers in your team in optimizing the queries. I also talked about the built-in system_health session trace that might be useful in RCA troubleshooting and collecting deadlock graphs out of the box. And last, but definitely not least, I discussed perfMon traces that are a must when collecting performance data of the entire system over a period of time.

SQL Server 2019 Improvements

Database and Server Upgrade

Hello. I'm Viktor Suha, and welcome back to this course, Managing SQL Server Database Performance. Having talked about the various aspects of database performance and management up until SQL Server of 2017, in this module I talk about SQL Server of 2019, specifically what's new in SQL Server 2019, what are the major performance-related improvements, and why you should consider upgrading if you haven't done so yet. I highlight only the most interesting new capabilities out of the many new features available in this release, focusing on performance. I discuss the on-premises or boxed version that you can install on an Azure virtual machine too. Some of the features appear differently in the Azure managed SQL Server offerings, Azure SQL Database and Azure SQL Managed Instance. I will note those differences separately. At some point in a database application lifecycle, the database will most probably be upgraded and migrated onto a newer database platform, like SQL Server 2019. The drivers for the upgrade can be different. In an enterprise environment, these factors are typically supportability, security, performance, and the advantage of using new product features. Your goal could be to leverage features that make your database application faster out of the box without the need to modify the code.

Versions, Editions, and Patching

I talk about product versions, editions, and patching in SQL Server 2019. Continuing the SQL Server version history, SQL Server 2019 has a major version of 15. In this timeline, SQL Server 2012, 2014, and 2016 are already in an extended support phase as of speaking in August 2021. SQL Server 2017 and 2019 are still in mainstream support according to the SQL Server product lifecycle. A new major version comes with a new database compatibility level 2. SQL Server 2019 introduced the database compatibility level 150 while still supporting older levels down to 100, which is the SQL Server 2008 level. The database compatibility level setting of your database is paramount as it determines the workload behavior and performance, whether new and improved features are used or not. Adding to the new performance-related features in database compatibility level 140 released with SQL Server 2017, database compatibility level 150 in SQL Server 2019 takes it even further. Previously, SQL Server 2017 introduced automatic tuning, adaptive joins for batch mode, interleaved execution for multi-statement table valued functions, and memory grant feedback for batch mode. SQL Server 2019 adds batch mode on a rowstore, memory grant feedback for row mode, scalar UDF inlining, and table variable deferred compilation. These are features that may make your database performance faster out of the box. I talk about these new features, called intelligent query processing features, in

more detail soon. SQL Server editions play a key role in developing your application and choosing the database platform. Why does the edition matter? Database features may be limited or not available at all in certain editions. Each edition may scale differently and have different resource utilization limits. And last, but definitely not least, editions have different price tags. The SQL Server edition and feature matrix has been updated with SQL Server 2019 specific information. You can find the details at the bit.ly link here or later in the references. Scale and resource utilization limits more or less remain the same; however, most of the new performance-related features under the moniker intelligent database tend to prefer the higher-end editions like the Enterprise edition. You can evaluate all those features in your development environment with the Free developer edition, which is Enterprise edition equivalent. The servicing model for SQL Server 2019 is the modern servicing model introduced with SQL Server 2017. There are no service packs released from SQL Server 2017 and onwards. Instead, you get cumulative update, or CU, packages released more frequently, and these packages may contain new feature additions too besides the bug fixes. As of speaking, the latest CU package for SQL Server 2019 is CU 12 released in August 2021. You can find all the SQL Server 2019 build versions and the related KB articles later in the references

Setup

I'll talk about the new SQL Server 2019 setup experience. This is another continuing trend as part of product improvements. It started with SQL Server 2016 where the setup helped you to configure tempdb in order to prevent tempdb contention and optimized tempdb performance in general. It was improved further in SQL Server 2017 while SQL Server 2019 adds new features to configure a few server configuration options adhering to the hardware you install based on configuration best practices. These server configuration options have direct impact on server performance. Let's see what these new setup options are. Proper memory configuration for your instance is paramount. SQL Server used default and highly nonoptimal, or rather not configured settings in previous versions that you had to manually adjust after installing the server, but definitely before going into production. In the new setup, the installer recommends min and max server memory settings that you can accept. These recommendations follow memory configuration best practices or guidelines. You can still use the default values or configure these settings manually, but it's much better to start with something more valid than to leave the defaults on. It's one of the most common SQL Server administration mistakes that people simply forget configuring the SQL Server memory settings. Similarly, MAXDOP, or max degree of parallelism, at the server level can be tricky to configure manually as the optimal setting highly depends on your hardware. The new setup recommends a MAXDOP value according to configuration guidelines; however, you can still opt out and configure it manually. Of course, you can specify these new setup options from the command prompt too when doing an unattended install. If you're using Enterprise Server CAL licensing, the setup can now send a warning if you have more than 20 physical CPU cores or 40 cores with hyperthreading enabled. This is another useful feature that can prevent a bad server configuration.

Tempdb

I'll talk about the new tempdb improvements in SQL Server 2019. Configuring tempdb and native tempdb performance was always the focus when optimizing SQL Server and work performance in general. As we move ahead in the timeline, the first major overhaul of how tempdb works and how it should be configured was done in SQL Server 2016. It introduced major built-in improvements that were only possible with trace flags previously, as well as the setup health and configuring the physical tempdb layout properly. It was further improved in SQL Server 2017, and now in SQL Server 2019, yet again, you can expect better built-in tempdb performance, plus there is a new improvement that you can choose to enable. You definitely get better tempdb performance out of the box with more recent versions, especially with SQL Server 2019. If you look back, the SQL Server 2016 setup installer helped you in configuring multiple tempdb data files with the proper file sizes and auto growth values that together with the trace flag 1118 and 1117 behaviors potentially mitigated common tempdb contention bottlenecks. Setup intelligence was further improved in SQL Server 2017 where you could specify very large tempdb data file sizes, and it also warned you if instant file initialization was not enabled. Taking it further, all databases, user and system databases alike, may benefit from further built-in metadata management improvements in SQL Server 2019. However, for tempdb, there still can be scenarios when you have certain metadata contention. SQL Server 2019 introduces a new server configuration setting called tempdb metadata memory-optimized. This is an Enterprise edition-only feature currently and is disabled by default in the on-premises version of SQL Server. If you experience specific tempdb metadata contention that often manifests itself as prevalent PAGELATCH waits in tempdb, you can turn this new option on and SQL Server will move the tempdb metadata in memory managed as in-memory OLTP. You can use the ALTER SERVER CONFIGURATION statement and set the MEMORY_OPTIMIZED TEMPDB_METADATA setting on. You'll need a service restart to take effect. If you want to verify if tempdb in-memory metadata is turned on or not, you can use a new server property called IsTempdbMetadataMemoryOptimized. Please note nothing comes completely free. Besides the service restart requirements, this feature currently has a few limitations and restrictions that you need to first evaluate and test with your particular workload.

Query Store

I talk about the new Query Store improvements in SQL Server 2019. The Query Store might be one of the most underutilized features in SQL Server. It's available since SQL Server 2016, and if you enable it on your database, you can obtain valuable work with performance metrics that can help you in both troubleshooting and query optimization. Moreover, many SQL Server environments are simply not monitored, so having a built-in feature like the Query Store can be very helpful. A big plus, it's available in all editions of SQL Server. Taking the Query Store features further, SQL Server 2019 introduces custom QUERY_CAPTURE_POLICY. There are two new options, QUERY_CAPTURE_MODE that when set to custom you can then specify your custom QUERY_CAPTURE_POLICY

with the QUERY_CAPTURE_POLICY option. This way, you can further optimize what you capture in the Query Store. That also means you can further optimize Query Store management in general. When turning the Query Store on for your database, you can specify multiple options that determine how the Query Story will function. This example uses the new custom value set for the QUERY_CAPTURE_MODE option, then it uses a custom capture policy specified with the QUERY_CAPTURE_POLICY option.

Accelerated Database Recovery

I talk about a new feature called accelerated database recovery in SQL Server 2019. If you have been working with SQL Server for a long time, you may have encountered problems with the way the standard database recovery process works. For example, the database recovery takes a long time to complete, or if you have a large transaction rollback it takes ages to finish. Database recovery relies on the transaction log, and the log truncation mechanism could result in an ever-growing transaction log file in case you have a long running transaction. The database recovery bottlenecks and how the log is managed may have a negative impact on database availability in general. To address these problems, SQL Server 2019 introduces a new overhauled database recovery process called accelerated database recovery. This is a feature that is available in both the Enterprise and Standard editions, but not in Express. It is disabled by default in the on-premises version of SQL Server. This is also an example for a feature that works differently in Azure SQL Database. In Azure SQL Database, it is enabled by default and you cannot even turn it off. You can enable accelerated database recovery Y on a per database level. There's a new database setting called ACCELERATED_DATABASE_RECOVERY that you can turn on with an ALTER DATABASE statement. If you want to verify if the setting is enabled on your databases, use the sys.databases catalog view and query the is_accelerated_database_recovery_on setting value. The advantages of this feature are clear. You can achieve a faster database recovery performance in general and a faster transaction rollback performance. The transaction log is truncated more aggressively. All in all, this feature can significantly improve database availability.

Sequential Key Insert Optimization

I talk about the new feature to optimize sequential key inserts in SQL Server 2019. When you have an index that uses an index key that is sequential, like an identity column, in case many threats try to insert new values into the index structure in peril, you can experience a common bottleneck called last page hotspot or last page insert contention when the newly added rows are added to the last index page. Concurrent threads need to wait for the page to be available while others finish first. You can commonly see prevalent PAGELATCH_EX waits that can become a bottleneck. To address this type of problem, SQL Server 2019 introduces a new index option called OPTIMIZE_FOR_SEQUENTIAL_KEY that you can specify on your indexes. This example shows how to use this new index option in line when creating a new table and specifying a primary key constraint that subsequently creates

an underlying clustered index. You can use the WITH clause to specify various index options, including the new OPTIMIZE_FOR_SEQUENTIAL_KEY option. Similarly, you can use this new index option when creating your indexes explicitly with the create index or altering existing indexes with the alter index statements.

Intelligent Query Processing

I talk about the new intelligent query processing features in SQL Server 2019. There has been query processing-related improvements in recent versions of SQL Server. New features, marked with blue in this diagram, were introduced in SQL Server 2017. New features of the intelligent query processing features, marked with green, were introduced in SQL Server 2019. These features are now available in Azure SQL Database and Azure SQL Managed Instance too. These are features that might be used by the query optimizer if all the prerequisites are met. These features may make your query workload faster out of the box, but in order to trigger this, you mostly need to run with the latest database compatibility level of 150, may need to use a higher-end edition, like the Enterprise edition, and sometimes your query code needs to adhere to certain requirements too. Let's see some of these one by one. Batch mode on rowstore can be a huge improvement for your existing workload. Especially as the name suggests, batch mode processing does not require column store indexing anymore. Column store indexing can come with further restrictions or simply you don't use complex analytical queries that would justify the use of column store indexes while you still want to benefit from batch mode execution by reducing the CPU footprint of your eligible queries. It's an Enterprise edition-only feature. Memory grant feedback for row mode takes memory feedback further. In SQL Server 2017, this was only available for batch mode, now row mode execution can also benefit from improved performance by optimizing memory usage of the affected query. It's an Enterprise edition-only feature again. Scalar UDF inlining is a very important feature as the performance bottlenecks imposed by scalar are user-defined functions in earlier versions of SQL Server can be a huge overhead. Now, there's a chance that the query optimizer can optimize the scalar UDF as part of the core query. However, this does not come free either. You need to adhere to many code-level prerequisites so that your functions can be inline. Plus, recent CU packages may alter even the existing behavior. You definitely need thorough testing to use this feature, but when it is triggered it can make your query performance match faster. Good thing it's available in all editions of SQL Server. Table variable deferred compilation impacts the optimization of table variables that, similar to scalar UDFs, were long considered a bad practice in general with previous versions of SQL Server. Now the query optimizer has better statistics and uses the actual cardinality of the table variable. As a result, you can have better query plans involving table variables. Good thing it's available in all editions of SQL Server. And an additional one. When working with really large data and you want to count the values, running a full-scale count might not be the best approach, especially if you just want an approximate count value. To address this problem, SQL Server 2019 introduces a new function called APPROX_COUNT_DISTINCT that accepts an expression and returns the approximate number of unique non-null values in a set. This is best used with millions of rows where the data values are mostly distinct. Then this approach

may scale better with a lower memory footprint than a traditional precise count. Good thing it's available in all editions of SQL Server.

References

There is so much more behind each of these features. You can explore how scalar UDF inlining can work in more detail in my other course, Troubleshooting SQL Server Performance Problems, where you can see this in action in a real-life scenario. Please refer to the following KB articles and documentation for more details on some of the main topics in this module. The SQL Server 2019 Build Versions KB article contains all the build versions that have been released since the RTM version, incremented mainly with CU packages. The SQL Server Modern Servicing Model KB article discusses this new servicing model, focusing on CU packages that you must apply in your SQL Server 2019 environment regularly. The SQL Server 2019 Editions and Features matrix shows which features are available in which edition This should be one of your first go-to documentations when planning a database migration or planning a new database application. The SQL Server 2019 Intelligent Query Processing documentation details each new feature, what are the prerequisites, and how you can manage them.

Module and Course Summary

That's a wrap for this module. I highlighted major new improvements and features in SQL Server 2019 that may have a positive impact on your database performance. Once you upgrade your database, there is chance that the new features may make your query workload faster out of the box. I talked about the new server major version, as well as the new database compatibility level additions and patching. I discussed the new set of features to help you in configuring domain Mac Server memory and the maxed observer configuration settings according to guidelines and best practices. I talked about tempdb improvements in particular and noted that overall tempdb performance has been greatly improved in SQL Server 2019, and you can even take it further by making the tempdb metadata memory optimized. If you don't use the Query Store yet, I encouraged you to use it as it provides a useful built-in mechanism to capture your queries so that you can analyze and troubleshoot your query performance. Now in SQL Server 2019, you can further optimize the Query Store with a custom capture policy. I gave you an introduction on the new and overhauled database recovery process, good accelerated database recovery, which can greatly improve your database availability, especially if you have already encountered bottlenecks, like overly long transaction rollbacks. This is a feature that is enabled by default in Azure SQL Database. You can opt-in manually in your on-premises version of SQL Server on a per-database level. You can optimize your indexes that suffer from the last page hotspot or last page insert contention problems by using a new index option called optimize for sequential key. This can be used when creating or altering your indexes. And finally, I listed the main intelligent query processing features that further enhanced what were introduced in SQL Server 2017. These are batch mode and rowstore,

memory grant feedback for row mode, scalar UDF inlining, and table variable deferred compilation. There is a new function called APPROX_COUNT_DISTINCT that you can use to do an approximate count in a very large dataset. I hope you like this overview of these great new features in SQL Server 2019. And we have also completed the course. It has been a long journey, but there is so much more. What we have covered in this course are the following: how to approach performance and scalability problems, what are the main areas to consider when working with SQL Server, also what does performance and scalability mean in the first place, why SQL Server versions, editions, patching, maintenance, and configuration matter when talking about performance and scalability, how some of the key components work in SQL Server under the cover so that they can be better configured, why server health checks are crucial for preventing major performance problems from occurring, how to configure important server configuration options like memory and parallelism-related settings, along with how to configure tempdb properly, how to size and choose service tiers for Infrastructure as a Service and Platform as a Service SQL Server offerings, what troubleshooting methods and tools are available in SQL Server, including DMVs for wait statistics analysis and the Query Store. Hope you found this course useful, and see you next time.