O'REILLY®

SQL Server Advanced Troubleshooting and Performance Tuning

Best Practices and Techniques

Early Release

RAW & UNEDITED

Dmitri Korotkevitch

SQL Server Advanced Troubleshooting and Performance Tuning

Best Practices and Techniques

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Dmitri Korotkevitch

SQL Server Advanced Troubleshooting and Performance Tuning

by Dmitri Korotkevitch

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Sarah Grey and Andy Kwan

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

April 2022: First Edition

Revision History for the Early Release

- 2020-12-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098101923 for release details.

978-1-098-10186-2

[LSI]

Chapter 1. SQL Server Setup and Configuration

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 1 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at dmitri@aboutsqlserver.com.

Database servers never live in a vacuum. They belong to an ecosystem of one or more applications used by customers. Application databases are hosted on one or more instances of SQL Server, which, in turn, run on physical or virtual hardware. The data is stored on disks that are usually shared with other customers and database systems. Finally, all components use a network for communication and storage.

The complexity and internal dependencies of database ecosystems make troubleshooting a very challenging task. From the customers' standpoint, most problems present themselves as general performance issues: applications might feel slow and unresponsive, database queries might time out, or applications might not connect to the database. The root cause of the issues could be anywhere. Hardware could be malfunctioning or incorrectly configured; the database might have inefficient schema, indexing, or code; SQL Server could be overloaded; client applications could have bugs or

design issues. This means you'll need to take a holistic view of your entire system in order to identify and fix problems.

This book is about troubleshooting SQL Server issues. However, we will always start this by analyzing your application ecosystem and SQL Server environment. This chapter will give you a set of guidelines on how to perform that validation and detect most common inefficiencies in SQL Server configuration. First, I'll discuss the hardware and operating system setup. Next, I'll talk about SQL Server and database configuration. I'll also touch on the topic of SQL Server consolidation and the overhead that monitoring can introduce into the system.

Hardware and Operating System Considerations

In most cases, troubleshooting and performance-tuning processes happen in production systems that host a lot of data and work under heavy loads. You have to deal with the issues and tune the live systems. Nevertheless, it is impossible to completely avoid discussion about hardware provisioning, especially because you may find that your servers cannot keep up with the load and need to be upgraded after troubleshooting.

I am not going to recommend particular vendors, parts or model numbers; computer hardware improves quickly and any such specific advice would be obsolete by the time the book is published. Instead, I'll focus on common-sense considerations with long-term relevance.

CPU

The license cost of a commercial database engine is, by far, the most expensive part in the system, and SQL Server is not an exception: you could build a decent server for less than the retail price of four cores in Enterprise Edition. You should buy the most powerful CPU your budget allows, especially if you are using non-Enterprise Editions, which limit the number of cores you can utilize.

Pay attention to CPU model. Each generation of CPUs will introduce performance improvements over the previous ones. You may get 10% to 15% performance improvements just by choosing newer CPUs, even when both generation of CPUs have the same clock speed.

In some cases, when licensing cost is not an issue, you may need to choose between slower CPUs with more cores and faster CPUs with fewer cores. In that case, the choice greatly depends on the system workload. In general, Online Transactional Processing (OLTP) systems, and especially In-Memory OLTP, would benefit from the higher single-core performance. A data warehouse and analytical workload, on the other hand, may run better with higher degree of parallelism and more cores.

Memory

There is a joke in the SQL Server community that goes like this:

Q. *How much memory does SQL Server usually need*?

A. *More*.

This joke has merits. SQL Server benefits from a large amount of memory, which allows it to cache more data. This, in turn, will reduce amount of disk input/output (I/O) activity and improve SQL Server's performance. Therefore, adding more memory to the server may be the cheapest and fastest way to address some performance issues.

For example, suppose the system suffers from non-optimized queries. You could reduce their impact by adding memory and eliminating the physical reads they introduce. This, obviously, does not solve the root cause of the problem. It is also dangerous, because as the data grows it eventually may not fit into the cache. However, in some cases it may be acceptable as a temporary Band-Aid solution.

The Enterprise Edition of SQL Server does not limit the amount of memory it can utilize. Non-Enterprise editions have limitations. In terms of memory utilization, Standard Edition of SQL Server 2016 and above can use up to 128GB of RAM for the buffer pool, 32GB of RAM per database for In-

Memory OLTP data, and 32GB of RAM for storing columnstore index segments. Web Edition memory usage is limited to half of what the Standard Edition provides. Factor those limits into your analysis when you are provisioning or upgrading non-Enterprise Edition instances of SQL Server. Don't forget to allocate some additional memory to other SQL Server components, such as plan cache and lock manager.

In the end, add as much memory as you can afford. It is cheap nowadays. There is no need to overallocate memory if your databases are small, but think about future data growth.

Disk Subsystem

A healthy, fast disk subsystem is essential for good SQL Server performance. SQL Server is very I/O intensive application - it is constantly reading from and writing data to disk.

There are many options for architecting the disk subsystem for SQL Server installations. The key is to build it in a way that provides low latency for I/O requests. For critical tier-1 systems, I recommend not exceeding 3 to 5 milliseconds (ms) of latency for data files reads and writes, and 1 to 2 ms of latency for transaction log writes. Fortunately, those numbers are now easily achieved with flash-based storage.

There's a catch, though: When you troubleshoot I/O performance in SQL Server, you need to analyze the latency metrics *within* SQL Server rather than on the storage level. It is common to see significantly higher numbers in SQL Server rather than in storage key performance indicators (KPIs), due to the queueing that may occur with I/O intensive workload. (Chapter 3 will discuss how to capture and analyze I/O performance data.)

If your storage subsystem provides multiple performance tiers, I recommend putting tempdb database on the fastest drive, followed by transaction log and data files. The tempdb is the shared resource on the server, and it is essential for this database to have good I/O throughput.

The writes to transaction log files are synchronous. It is critical to have low write latency for those files. The writes to transaction log are also sequential; however, remember that placing multiple log and/or data files to the same drive will lead to random I/O across multiple databases.

As a best practice, I'd put data and log files to the different physical drives for maintainability and recoverability reasons. You need to look at the underlying storage configuration though. In some cases, when disk arrays do not have enough spindles, splitting them across multiple LUNs may degrade disk array performance.

In my systems, I do not split clustered and nonclustered indexes across multiple filegroups by placing them on different drives. It rarely improves I/O performance unless you can completely separate storage across the filegroups. On the other hand, this configuration can significantly complicate disaster recovery.

Finally, remember that some SQL Server technologies benefit from good sequential I/O performance. For example, In-Memory OLTP does not use random I/O at all, and performance of sequential reads usually becomes the limiting factor for database startup and recovery. Data Warehouse scans would also benefit from sequential I/O performance when B-Tree and columnstore indexes are not heavily fragmented. The difference between sequential and random I/O performance is not very significant with flash-based storage; however, it may be a big factor with magnetic drives.

Network

SQL Server communicates with clients and other servers via the network. Obviously, it needs to provide enough bandwidth to support that communication. There are a couple items I want to mention, though.

First, you need to analyze entire network topology when you troubleshoot network-related performance. Remember that a network's throughput will be limited to the speed of its slowest component. You may have a 10 Gbps uplink from the server; however, if you have 1Gbps switch in network path,

that would become the limiting factor. This is especially critical for network-based storage: make sure that I/O path to disks is as efficient as possible.

Second, if you are using AlwaysOn Failover Cluster or AlwaysOn Availability Groups, create a separate network for the cluster heartbeat. In some cases, you may also consider building separate network for Availability Group traffic. Just make sure that those networks are properly utilized for cross-node communication.

Consider a situation where you have a virtualized SQL Server cluster with nodes running on different hosts. You would need to check that the hosts can separate and route the traffic in the cluster network separately from the client traffic. Serving all vLan traffic through the single physical network card would defeat the purpose of a heartbeat network. (I will talk more about troubleshooting network-related issues in Chapter 14.)

Operating Systems and Applications

As a general rule, I suggest using the most recent version of your operating system that supports your version of SQL Server. Make sure that both the OS and SQL Server are patched, and implement a process to do patching regularly.

Do *not* use the 32-bit version of SQL Server. The 64-bit version outperforms it and scales better with the hardware.

Since SQL Server 2017, it's been possible to use Linux to host the database server. From a performance standpoint, Windows and Linux versions of SQL Server are very similar. The choice of operating system depends on enterprise ecosystem and on what your team is more comfortable to support. Keep in mind, that Linux-based deployments may require a slightly different High Availability (HA) strategy compared to a Windows setup. For example, you may have to rely on Pacemaker instead of Windows Server Failover Cluster (WSFC) for automatic failovers.

Use a dedicated SQL Server host whenever possible. Remember that it's easier and cheaper to scale application servers—don't waste valuable

resources on the database host!

On the same note, do not run nonessential processes on the server. I see database engineers running SQL Server Management Studio (SSMS) in remote desktop sessions all the time. It is always better to work remotely and not consume server resources.

Finally, if you are required to run antivirus software on the server, exclude any database folders from the scan.

Virtualization and Clouds

Modern IT infrastructure depends heavily on virtualization, which provides additional flexibility, simplifies management, and reduces hardware costs. As a result, more often than not, you'll have to work with virtualized SQL Server infrastructure.

There is nothing wrong with that. Properly implemented virtualization gives you many benefits, with negligible performance overhead. It adds another layer of High Availability with vMotion or Live Migration. It allows you to seamlessly upgrade the hardware and simplifies database management. Unless you have the edge case when you need to squeeze the most from the hardware, I suggest virtualizing your SQL Server ecosystem.

Virtualization, however, adds another layer of complexity during troubleshooting. You need to pay attention to the host's health and load in addition to guest virtual machine (VM) metrics. To make matters worse, the performance impact of an overloaded host might not be clearly visible in standard performance metrics in guest OS.

I will discuss several approaches to troubleshooting the virtualization layer in Chapter 16; however, you can start by working with infrastructure engineers to confirm that the host is not overprovisioned. Pay attention to the number of physical CPUs and allocated vCPUs on the host along with physical and allocated memory. Mission-critical SQL Server VMs should have resources reserved for them to avoid performance impact.

Asides from the virtualization layer, troubleshooting virtualized SQL Server instances is the same as troubleshooting physical ones. The same applies to cloud installations when SQL Server is running within virtual machines. After all, the cloud is just a different datacenter managed by an external provider.

Configuring Your SQL Server

The SQL Server setup process's default configuration is relatively decent and may be suited to light and even moderate workloads. There are several things you need to validate and tune, however.

SQL Server Version and Patching Level

SELECT @@VERSION is the first statement I run during SQL Server system health checks. There are two reasons for that. First, it gives me a glimpse of the system's patching strategy, so I can potentially suggest some improvements. Second, it helps me to identify possible known issues that may exist in the system.

The latter one is very important. Many times, customers have asked me to troubleshoot problems that had already been resolved by service packs and cumulative updates. Always look at the release notes to see if any of the issues mentioned look familiar; your problem may have already been fixed.

You might consider upgrading to the newest version of SQL Server when possible. Each version introduces performance, functional and scalability enhancements. This is especially true if you move to SQL Server 2016 or above from older versions. SQL Server 2016 was a milestone release that included many performance enhancements. In my personal experience, upgrading from SQL Server 2012 to 2016 and above can improve performance by 20 to 40% without any additional steps.

It is also worth noting that starting with SQL Server 2016 SP1, the former Enterprise Edition-only features became available in the lower-end editions

of the product. Some of them, like data compression, allow SQL Server to cache more data in the buffer pool and improve performance of the system.

Obviously, you need to test the system prior to upgrading – there is always the chance of regressions. The risk is usually small with minor patching; however, it increases with the major upgrades. You can mitigate some risks with several database options, as you will see later in this chapter.

Instant File Initialization

Every time SQL Server grows data and transaction log files—either automatically or as part of ALTER DATABASE command—it fills the newly allocated part of the file with zeros. This process blocks all sessions that are trying to write to the corresponding file and, in case of transaction log, stops generating any log records. It may also generate the spike in I/O write workload.

That behavior cannot be changed for transaction log files – SQL Server always zeros them out. However, you can disable it for the data files by enabling *instant file initialization (IFI)*. This speeds up data file growth and reduces the time required to create or restore databases.

You can enable instant file initialization by giving an SA_MANAGE_VOLUME_NAME permission, also known as *Perform Volume Maintenance Task*, to the SQL Server startup account. This can be done in the *Local Security Policy* management application (secpol.msc). You will need to restart SQL Server for the change to take effect.

In SQL Server 2016 and above, you can also grant this permission as part of the SQL Server setup process (shown in Figure 1-1).

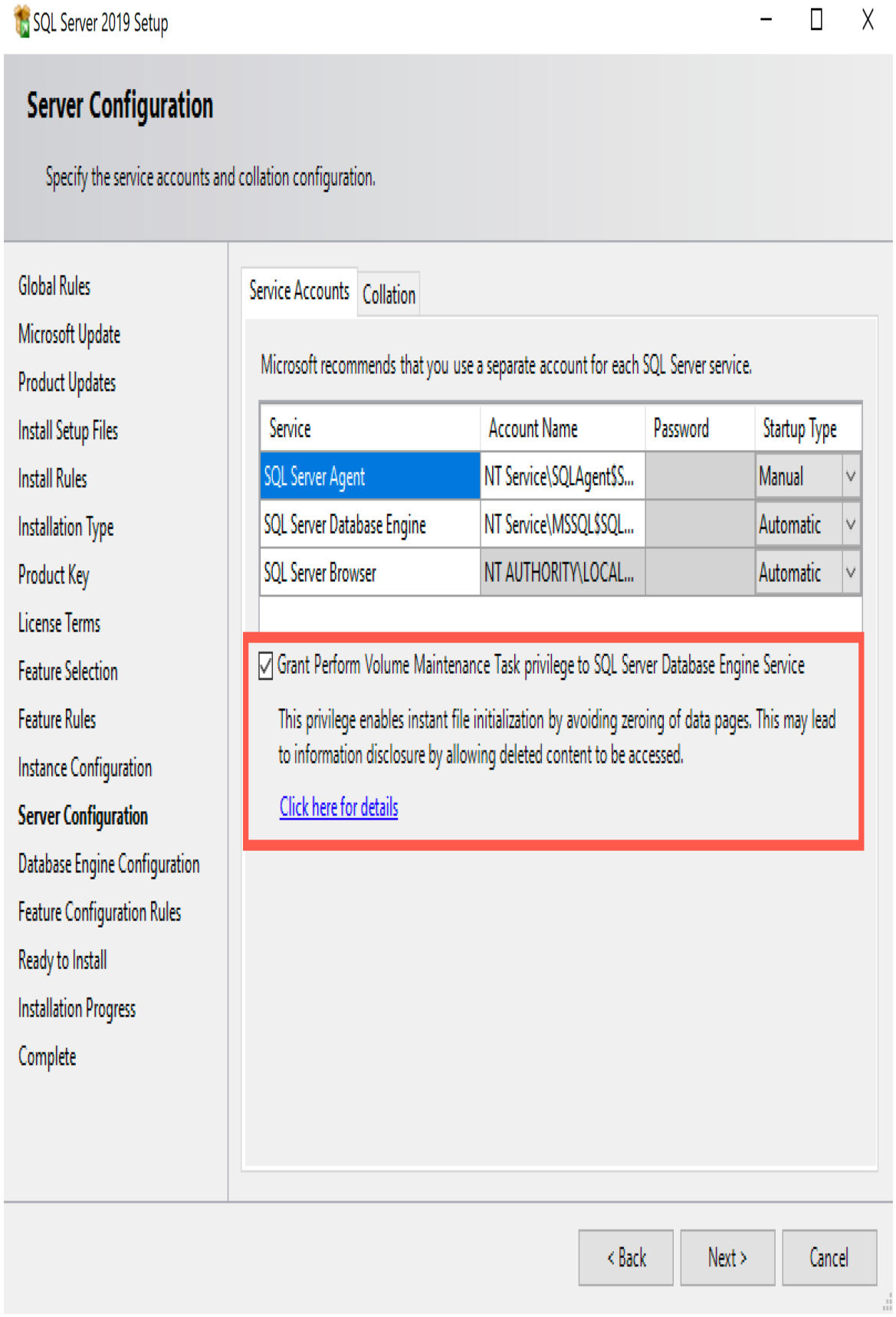
SQL Server 2019 Setup
Server Configuration
Specify the service accounts and collation configuration.
Global Rules
Microsoft Update
Product Updates
Install Setup Files
Install Rules
Installation Type
Product Key
License Terms
Feature Selection
Feature Rules
Instance Configuration
Server Configuration
Database Engine Configuration
Feature Configuration Rules
Ready to Install
Installation Progress
Complete
Service Accounts
Collation
Microsoft recommends that you use a separate account for each SQL Server service.
Service
Account Name
Password
Startup Type
SQL Server Agent
NT Service\SQLAgent$S...
Manual
SQL Server Database Engine
NT Service\MSSQL$SQL...
Automatic
SQL Server Browser
NT AUTHORITY\LOCAL...
Automatic
Grant Perform Volume Maintenance Task privilege to SQL Server Database Engine Service
This privilege enables instant file initialization by avoiding zeroing of data pages. This may lead to information disclosure by allowing deleted content to be accessed.
Click here for details
< Back
Next >
Cancel

Figure 1-1. Enabling Instant File Initialization during SQL Server setup.

You can check if IFI is enabled by examining the instant_file_initialization_enabled column in the sys.dm_server_services data management view. This column is available in SQL Server 2012 SP4, SQL Server 2016 SP1, and above. In older versions, you can run the code shown in Listing 1-1.

Listing 1-1. Checking if instant file initialization is enabled

```
DBCC TRACEON(3004,3605,-1);
go
CREATE DATABASE Dummy;
go
EXEC sp_readerrorlog;
go
DROP DATABASE Dummy;
go
DBCC TRACEOFF(3004,3605,-1);
go
```

If IFI is not enabled, the SQL Server log will indicate that SQL Server is zeroing out the .mdf data file in addition to zeroing out the log .ldf file, as shown in Figure 1-2. When IFI is enabled, it will only show zeroing out of the log .ldf file.

	LogDate	ProcessInfo	Text
104	2020-10-26 15:57:45.370	spid32s	A connection timeout has occurred while attempting to establish a connection to availa…
105	2020-10-26 15:58:35.510	spid51	DBCC TRACEON 3004, server process ID (SPID) 51. This is an informational message only;…
106	2020-10-26 15:58:35.510	spid51	DBCC TRACEON 3605, server process ID (SPID) 51. This is an informational message only;…
107	2020-10-26 15:58:35.520	spid51	Zeroing C:\DB\Dummy.mdf from page 0 to 1024 (0x0 to 0x800000)
108	2020-10-26 15:58:35.530	spid51	Zeroing completed on C:\DB\Dummy.mdf (elapsed = 2 ms)
109	2020-10-26 15:58:35.530	spid51	Zeroing C:\DB\Dummy_log.ldf from page 0 to 1024 (0x0 to 0x800000)
110	2020-10-26 15:58:35.540	spid51	Zeroing completed on C:\DB\Dummy_log.ldf (elapsed = 2 ms)
111	2020-10-26 15:58:35.550	spid51	Starting up database 'Dummy'.

Figure 1-2. Checking if instant file initialization is enabled.

There is a small security risk associated with this setting. When IFI is enabled, the database administrators may see some data from previously deleted files in OS by looking at newly allocated data pages in the database. This is acceptable in most systems; if so, enable it.

Tempdb Configuration

Tempdb is the system database used to store temporary objects created by users and by SQL Server internally. This is a very active database and it often becomes a source of contention in the system. I will discuss how to troubleshoot tempdb-related issues in Chapter 10; in this chapter, I'll focus on configuration.

As already mentioned, you need to place tempdb on the fastest drive in the system. Generally speaking, this drive does not need to be redundant nor

persistent – the database is recreated at SQL Server startup, and local SSD disk or cloud ephemeral storage would work fine. Remember, however, that SQL Server will go down if tempdb is unavailable, so factor that into your design.

If you are using non-Enterprise Editions of SQL Server and the server has more memory than SQL Server can consume, you can put tempdb on the RAM drive. Don't do that with Enterprise Edition, though – you'll usually achieve better performance by using that memory for the buffer pool.

> **NOTE**
>
> Pre-allocate tempdb files to the maximum size of RAM drive and create additional small data and log files on disk to avoid running out of space. SQL Server will not use small on-disk files until RAM drive files are full.

The tempdb database should always have multiple data files. Unfortunately, default configuration created by SQL Server setup is not optimal, especially in the old versions of the product. We will discuss how to fine-tune the number of data files in tempdb later in the book, but you can use the following as the rule of thumb in initial configuration:

If the server has 8 or fewer CPU cores, create the same number of data files.

If server has more than 8 CPU cores, use either 8 data files or 1/4 of the number of cores, whichever is greater, rounding up in batches of 4 files. For example, use 8 data files in the 24-core server and 12 data files in the 40-core server.

Finally, make sure that all tempdb data files have the same initial size and auto-growth parameters specified in megabytes (MB) rather than in percentages. This will allow SQL Server to better balance the usage of the data files, reducing possible contention in the system.

Trace Flags

SQL Server uses trace flags to enable or change the behavior of some product features. Although Microsoft has introduced more and more database and server configuration options in new versions of SQL Server, trace flags are still widely used. You will need to check any trace flags that are present in the system; you may also need to enable some of them.

You can get the list of enabled trace flags by running the `DBCC TRACESTATUS` command. You can enable them in SQL Server Configuration Manager and/or by using `-T` SQL Server startup option.

Let's look at some common trace flags.

T1118

> This trace flag prevents usage of `mixed extents` in SQL Server. This will help to improve `tempdb` throughput in SQL Server 2014 and below by reducing the amount of changes and, therefore, contention in `tempdb` system catalogs. This trace flag is not required in SQL Server 2016 and above, where `tempdb` does not use mixed extents by default.

T1117

> With this trace flag, SQL Server auto-grows all data files in the filegroup when one of the files is out of space. It provides more balanced I/O distribution across data files. You should enable it to improve `tempdb` throughput in old versions of SQL Server; however, check if any users' databases have filegroups with multiple unevenly sized data files. As with `T1118`, this trace flag is not required in SQL Server 2016 and above, where `tempdb` auto-grows all data files by default.

T2371

> By default, SQL Server automatically updates statistics only after 20% of the data in the index has been changed. This means that with large tables, statistics are rarely updated automatically. The trace flag T2371 changes this behavior, making the statistics update threshold dynamic – the larger the table is, the lower the percentage of changes required to trigger the update. Starting with SQL Server 2016, you can also control this

behavior via database compatibility level. I recommend enabling this trace flag unless all databases on the server have a compatibility level of 130 or above.

T3226

With this trace flag, SQL Server does not write information about successful database backups to the error log. This may help to reduce the size of the logs, making them more manageable.

T1222

This trace flag writes deadlock graphs to the SQL Server error log. This flag is benign; however, it makes SQL Server log harder to read and parse. It is also redundant – you can get deadlock graph from `System_Health` Extended Event session when needed. I usually remove this trace flag when I see it.

T4199

This trace flag and the `QUERY_OPTIMIZER_HOTFIXES` database option (in SQL Server 2016 and above) control the behavior of Query Optimizer hotfixes. When enabled, the hotfixes introduced in service packs and cumulative updates will be used. This may help to address some of Query Optimizer bugs and improve query performance; however, it also increases the chance of plan regressions after patching. I usually do not enable it in production systems unless it is possible to perform thorough regression testing of the system before patching.

To summarize – in SQL Server 2014 and below, enable `T1118`, `T2371` and, potentially, `T1117`. In SQL Server 2016 and above, enable `T2371` unless all databases have compatibility level of 130 or above. After that – look at all other trace flags in the system and understand what they are doing. Some trace flags may be inadvertently installed by third-party tools and can negatively affect server performance.

Server Options

SQL Server provides many configuration settings. I'll cover many of them in depth later in the book; however, there are a few settings worth mentioning here.

Optimize for Ad hoc Workloads

The first one is *Optimize for Ad-hoc Workloads*. This configuration option controls how SQL Server caches execution plans of ad-hoc (non-parameterized) queries. When it is disabled (by default), SQL Server caches full execution plans of those statements, which may significantly increase plan cache memory usage. As the opposite, when this setting is enabled, SQL Server starts by caching the small structure (just a few hundred bytes) called *plan stub*, replacing it with the full execution plan if an ad hoc query is executed the second time.

In majority of the cases, ad-hoc statements are not executed repeatedly, and it is beneficial to enable Optimize for Ad-hoc Workloads setting in every system. It could significantly reduce plan cache memory usage at cost of infrequent additional recompilations of ad-hoc queries. Obviously, this setting would not affect caching behavior of parameterized queries and T-SQL database code.

> **NOTE**
>
> Starting with SQL Server 2017 and in Azure SQL Database, you can control *Optimize for Ad-hoc Workload* behavior on the database level with the `OPTIMIZE_FOR_AD_HOC_WORKLOADS` database scoped configuration.

Max Server Memory

The second important setting is *Max Server Memory*, which controls how much memory SQL Server can consume. Database engineers love to debate how to properly configure it, and there are different approaches to calculating the proper value for the setting. Many engineers even suggest

leaving the default value in place and allowing SQL Server to manage it automatically. In my opinion, it is best to fine-tune that setting, but it's important to do so correctly (Chapter 8 will discuss the details). An incorrect setting will impact SQL Server performance more than if you leave the default value in place.

One particular issue I often encounter during system health checks is severe underprovisioning of this setting. Sometimes people forget to change it after hardware or VM upgrades; other times, it's incorrectly calculated in nondedicated environments, where SQL Server is sharing the server with other applications. In both cases, you can get immediate improvements by increasing Max Server Memory or even setting it to the default value until you perform full analysis later.

Affinity Mask

You need to check SQL Server affinity and, potentially, set *affinity mask* if SQL Server is running on hardware with multiple non-uniform memory access (NUMA) nodes. In modern hardware, each physical CPU usually becomes a separate NUMA node. If you restrict SQL Server from using some of the physical cores, you need to balance SQL Server CPUs (or schedulers – see Chapter 2) evenly across NUMAs. For example, if you are running SQL Server on a server with two 18-core Xeon processors and limiting SQL Server to 24 cores, you need to set affinity mask to utilize 12 cores from each physical CPU. This will give you better performance than having SQL Server use 18 cores from the first processor and 6 cores from the second.

Listing 1-2 shows how to analyze SQL Server schedulers (CPUs) distribution between NUMA nodes. Look at the count of schedulers for each parent_node_id column in the output.

Example 1-2. Checking the distribution of NUMA node schedulers (CPUs)

```
SELECT
  parent_node_id
  ,COUNT(*) as [Schedulers]
  ,SUM(current_tasks_count) as [Current]
  ,SUM(runnable_tasks_count) as [Runnable]
FROM sys.dm_os_schedulers
```

```
WHERE status = 'VISIBLE ONLINE'
GROUP BY parent_node_id;
```

Parallelism

It is important to check parallelism settings in the system. Default settings, like MAXDOP = 0 and Cost Threshold for Parallelism = 5, do not work well in modern systems. As with Max Server Memory, it is better to fine-tune the settings based on the system workload (Chapter 7 will discuss this in detail). However, my rule of thumb for generic settings is:

- Set MAXDOP to 1/4 of the number of available CPUs in OLTP and half those in Data Warehouse systems. Do not exceed the number of CPUs in the NUMA node.
- Set Cost Threshold for Parallelism to 50.

Starting with SQL Server 2016 and in Azure SQL Server Database, you can set MAXDOP on the database level using the command ALTER DATABASE SCOPED CONFIGURATION SET MAXDOP. This is useful when the instance hosts databases that handle different workloads.

Configuration Settings

As with trace flags, analyze other changes in configuration settings that have been applied on the server. You can examine current configuration options using the sys.configurations view. Unfortunately, SQL Server does not provide a list of default configuration values to compare, so you need to hardcode it, as shown in Listing 1-3. I am including just a few configuration settings to save space, but you can download the full version of the script from this book's examples repository.

Listing 1-3. Detecting changes in server configuration settings.

```
DECLARE
    @defaults TABLE
    (
        name SYSNAME NOT NULL PRIMARY KEY,
        def_value SQL_VARIANT NOT NULL
    )
INSERT INTO @defaults(name,def_value) VALUES('backup compression
```

```
default',0);
INSERT INTO @defaults(name,def_value) VALUES('cost threshold for
parallelism',5);
INSERT INTO @defaults(name,def_value) VALUES('max degree of
parallelism',0);
INSERT INTO @defaults(name,def_value) VALUES('max server memory
(MB)',2147483647);
INSERT INTO @defaults(name,def_value) VALUES('optimize for ad hoc
workloads',0);
/* Other settings are ommited in the book */
SELECT
    c.name, c.description, c.value_in_use, c.value
    ,d.def_value, c.is_dynamic, c.is_advanced
FROM
    sys.configurations c JOIN @defaults d ON
        c.name = d.name
WHERE
    c.value_in_use <> d.def_value OR
    c.value <> d.def_value
ORDER BY
    c.name;
```

Figure 1-3 shows the sample output of the code. The discrepancy between value and value_in_use columns indicates pending configuration changes that require restart to take an effect. The is_dynamic column shows if configuration option can be modified without restart.

	name	description	value_in_use	value	def_value	is_dynamic	is_advanced
1	max degree of parallelism	maximum degree of paralle…	1	1	0	1	1
2	optimize for ad hoc workloads	When this option is set, …	1	1	0	1	1

Figure 1-3. Non-default server configuration options.

Configuring Your Databases

As the next step in analyzing your configuration, you'll need to validate several database settings and configuration options. Let's look at them.

Database Settings

SQL Server allows you to change multiple database settings, tuning behavior to workload and other requirements. I'll cover many of them later in the book; however, there are a few settings I would like to discuss here.

The first one is *Auto Shrink*. When this option is enabled, SQL Server periodically shrinks the database and releases unused free space from the files to the OS. While this looks appealing and promises to reduce disk space utilization, it may introduce issues.

Implementing this database shrink process, automatically or through the command DBCC SHRINKFILE, works on the physical level. It locates empty space in the beginning of the file and moves allocated extents from the end of the file there, without taking extent ownership into consideration. This introduces noticeable load and lead to excessive index fragmentation. What's more, in many cases it's useless: the database files simply expand again as the data grows. It's always better to manage file space manually and disable Auto Shrink.

Another database option, *Auto Close*, controls how SQL Server caches data from the database. When it's enabled, SQL Server removes data pages from the buffer pool and execution plans from the plan cache when the database does not have any active connections. This will lead to performance impact with the new workload when data needs to be cached and queries need to be compiled again.

With very few exceptions, you should disable Auto Close. One such exception may be an instance that hosts a large number of rarely accessed databases. Even then, I would consider keeping this option disabled and allowing SQL Server to retire cached data in the normal way.

Make sure that *Page Verify* option is set to CHECKSUM. This will detect consistency errors more efficiently and helps to resolve database corruption cases.

Pay attention to the *database recovery model*. If the databases are in SIMPLE mode, in case of disaster or human error it would be impossible to

recover the data beyond the last FULL database backup. If you find the database in this mode, immediately discuss it with the stakeholders, making sure that they understand the risk of data loss.

Database Compatibility Level controls SQL Server's compatibility and behavior on the database level. For example, if you are running SQL Server 2019 and have a database with a compatibility level of 130 (SQL Server 2016), SQL Server will behave as if the database is running on SQL Server 2016. Keeping the databases on the lower compatibility levels simplifies SQL Server upgrades by reducing possible regressions; however, it also blocks you from getting some new features and enhancements.

As a general rule, run databases on the latest compatibility level that matches the SQL Server version. Be careful when you change it: as with any version change, this may lead to regressions. Test the system before the change and make sure you can roll back the change if needed, especially if the database has a compatibility level of 110 (SQL Server 2012) or below. Increasing compatibility level to 120 (SQL Server 2014) or above will enable a new cardinality estimation model and may significantly change execution plans for the queries. Test the system thoroughly to understand the impact of the change.

You can force SQL Server to use legacy cardinality estimation models with the new database compatibility levels by setting LEGACY_CARDINALITY_ESTIMATION database option to ON in SQL Server 2016 and above, or by enabling server-level trace flag T9481 in SQL Server 2014. This approach will allow you to perform upgrade or compatibility level changes in phases, reducing impact to the system. (Chapter 5 will cover cardinality estimation in more detail.)

Transaction Log Settings

SQL Server uses *write-ahead logging*, persisting information about all database changes in a transaction log. SQL Server works with transaction logs sequentially, in merry-go-round fashion. In most cases, you won't need

multiple log files in the system – they make database administration more complicated and do not improve performance.

Internally, SQL Server splits transaction logs into chunks called *Virtual Log Files (VLF)* and manages them as single units. For example, SQL Server cannot truncate and reuse a VLF if it contains just a single active log record. Pay attention to the number of VLFs in the database. Too few of them will lead to very large VLFs, which make log management and truncation suboptimal. Too many small VLFs will degrade the performance of transaction log operations. Try not to exceed several hundred VLFs in production systems.

SQL Server adds 16 VLFs every time it grows a transaction log. Unfortunately, the default 10% auto-growth configuration generates lots of unevenly sized VLFs. Change the log auto-growth setting to grow the file in chunks – I usually use chunks of 1,024 MB.

You can count the VLFs in the database by running DBCC LOGINFO. If the transaction log isn't configured well, consider rebuilding it. You can do this by shrinking the log to the minimal size and growing it in chunks of 1,024MB to 4,096 MB.

Do not auto-shrink transaction log files. They will grow again and affect performance when SQL Server zeroes out the file. It is better to pre-allocate the space and manage log file size manually. Do not restrict the maximum size and auto-growth, though – you want logs to grow automatically in case of emergencies. (Chapter 11 will provide more details on how to troubleshoot transaction-log issues.)

Data Files and Filegroups

By default, SQL Server creates new databases using the single-file PRIMARY filegroup and one transaction log file. Unfortunately, this configuration is suboptimal from performance, database management and High Availability standpoints.

SQL Server tracks space usage in the data files through system pages called *allocation maps*. In systems with highly volatile data, allocation maps can be a source of contention: SQL Server serializes access to them during their modifications (more about this in Chapter 10). Each data file has its own set of allocation map pages and you can reduce contention by creating multiple files in the filegroup with the active modifiable data.

Ensure that data is evenly distributed across multiple data files in the same filegroup. SQL Server uses an algorithm called *Proportional Fill*, which writes data to the file that has the most free space. Evenly sized data files will help to balance those writes, reducing allocation maps contention. Make sure that all data files in the filegroup have the same size and auto-growth parameters, specified in MBs.

You may also want to enable the AUTOGROW_ALL_FILES filegroup option (available in SQL Server 2016 and above), which triggers auto-growth for all files in the filegroup simultaneously. You can use trace flag T1117 for this in prior versions of SQL Server, but remember that this flag is set on the server level and will affect all databases and filegroups in the system.

It is often impractical or impossible to change the layout of existing databases. However, you may need to create new filegroups and move data around during performance tuning. Here are a few suggestions for doing so efficiently:

- Create multiple data files in filegroups with volatile data. I usually start with four files and increase the number if I see latching issues (see Chapter 10). Make sure that all data files have the same size and auto-growth parameters specified in MB; enable the AUTOGROW_ALL_FILES option. For filegroups with read-only data, one data file is usually enough.
- Do not spread clustered indexes, and nonclustered indexes, or large object (LOB) data across multiple filegroups. This rarely helps with performance and may introduce issues in cases of database corruption.

- Place related entities (for example, Orders and OrderLineItems) in the same filegroup. This will simplify database management and disaster recovery.
- Keep the PRIMARY filegroup empty if possible.

Figure 1-4 shows an example of a database layout for a hypothetical shopping-cart system. The data is partitioned and spread across multiple filegroups with the goal of minimizing downtime and utilizing *partial database availability* in case of disaster.[1] It will also improve your backup strategy by implementing partial database backups and excluding read-only data from FULL backups.

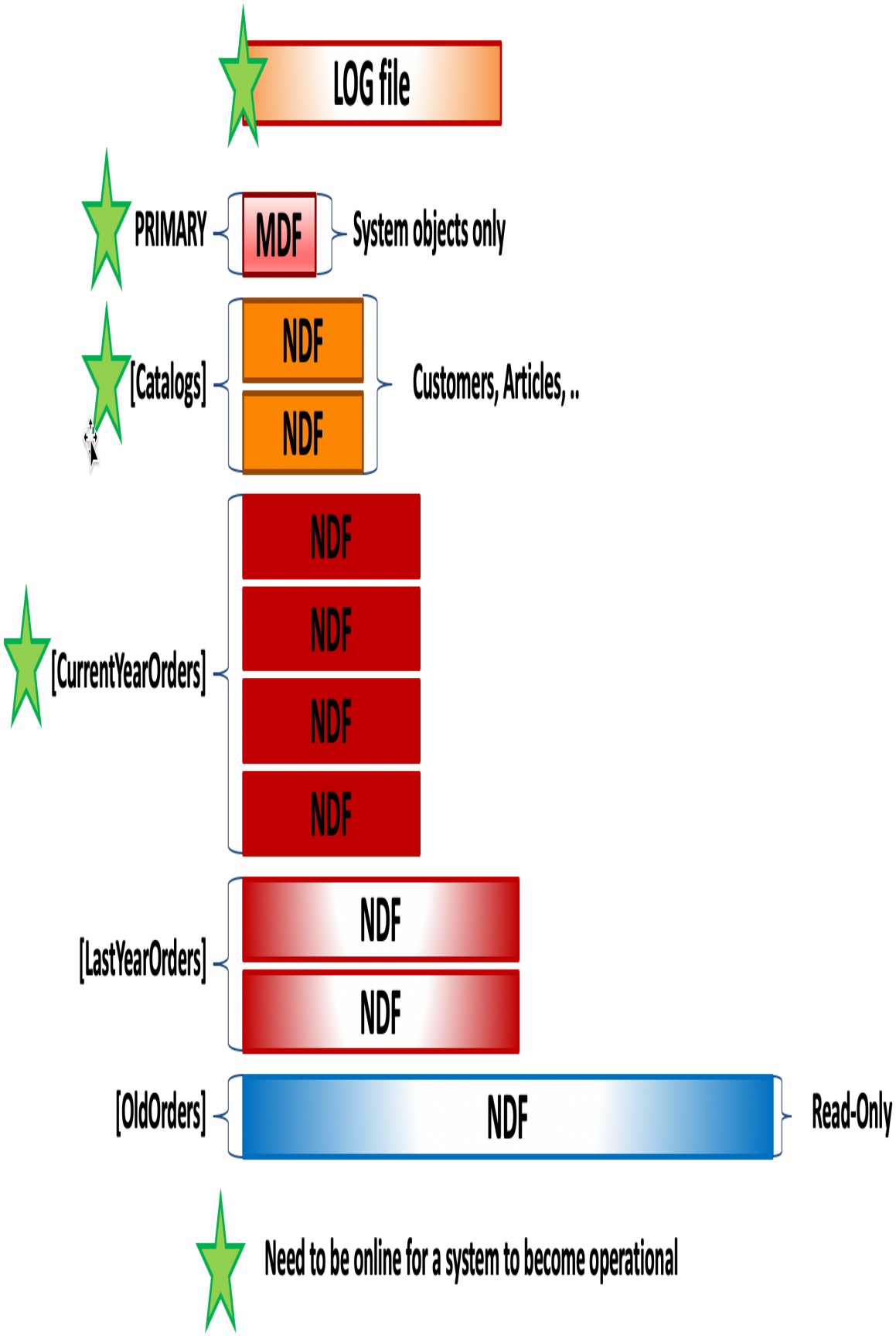
LOG file
PRIMARY
MDF
System objects only
[Catalogs]
NDF
NDF
Customers, Articles, ..
[CurrentYearOrders]
NDF
NDF
NDF
NDF
[LastYearOrders]
NDF
NDF
[OldOrders]
NDF
Read-Only
Need to be online for a system to become operational

Figure 1-4. Database layout for a shopping cart system.

Analyzing SQL Server Log

SQL Server Log is another place I usually check at the beginning of troubleshooting. I like to see any errors it has, which can point to some areas to follow up. For example, errors 823 and 824 can indicate issues with disk subsystem and/or database corruption.

You can read the content of the log in SSMS. You can also get it programmatically using the xp_readerrorlog system stored procedure. The challenge here is the amount of data in the log: the noise from the information messages may hide useful data.

The code in Listing 1-4 helps you to address that problem. It allows you to filter out unnecessary noise and focus on the error messages. You can control the behavior of the code with the following variables:

`@StartDate` and `@EndDate`

: Define the time for analysis.

`@NumErrorLogs`

: Specifies the number of log files to read if SQL Server rolls them over.

`@ExcludeLogonErrors`

: Omits logon auditing messages.

`@ShowSurroundingEvents` and `@ExcludeLogonSurroundingEvents`

: These allow you to retrieve the information messages around the error entries from the log. The time window for those messages is controlled by the `@SurroundingEventsBeforeSeconds` and `@SurroundingEventsAfterSeconds` variables.

Example 1-4. Listing 1-4. Analyzing SQL Server Error Log

```
IF OBJECT_ID('tempdb..#Logs',N'U') IS NOT NULL DROP TABLE #Logs;
IF OBJECT_ID('tempdb..#Errors',N'U') IS NOT NULL DROP TABLE
#Errors;
go
CREATE TABLE #Errors
(
  LogNum INT NULL,
  LogDate DATETIME NULL,
  ID INT NOT NULL identity(1,1),
  ProcessInfo VARCHAR(50) NULL,
  [Text] VARCHAR(MAX) NULL,
  PRIMARY KEY(ID)
);
CREATE TABLE #Logs
(
  [LogDate] DATETIME NULL,
  ProcessInfo VARCHAR(50) NULL,
  [Text] VARCHAR(max) NULL
);
DECLARE
  @StartDate DATETIME = DATEADD(DAY,-7,GETDATE())
  ,@EndDate DATETIME = GETDATE()
  ,@NumErrorLogs INT = 1
  ,@ExcludeLogonErrors BIT = 1
  ,@ShowSurroundingEvents BIT = 1
  ,@ExcludeLogonSurroundingEvents BIT = 1
  ,@SurroundingEventsBeforeSecond INT = 5
  ,@SurroundingEventsAfterSecond INT = 5
  ,@LogNum INT = 0;
WHILE (@LogNum <= @NumErrorLogs)
BEGIN
  INSERT INTO #Errors(LogDate,ProcessInfo,Text)
    EXEC [master].[dbo].[xp_readerrorlog]
      @LogNum, 1, N'error', NULL, @StartDate, @EndDate, N'desc';
  IF @@ROWCOUNT > 0
    UPDATE #Errors SET LogNum = @LogNum WHERE LogNum IS NULL;
  SET @LogNum += 1;
END;
IF @ExcludeLogonErrors = 1
  DELETE FROM #Errors WHERE ProcessInfo = 'Logon';
-- Errors only
SELECT * FROM #Errors ORDER BY LogDate DESC;
IF @@ROWCOUNT > 0 AND @ShowSurroundingEvents = 1
BEGIN
  DECLARE
    @LogDate DATETIME
```

```
    ,@ID INT = 0
  WHILE 1 = 1
  BEGIN
    SELECT TOP 1 @LogNum = LogNum, @LogDate = LogDate, @ID = ID
    FROM #Errors
    WHERE ID > @ID
    ORDER BY ID;
    IF @@ROWCOUNT = 0
      BREAK;
    SELECT
      @StartDate = DATEADD(SECOND, -@SurroundingEventsBeforeSecond,
@LogDate)
      ,@EndDate = DATEADD(SECONd, @SurroundingEventsAfterSecond,
@LogDate);
    INSERT INTO #Logs(LogDate,ProcessInfo,Text)
      EXEC [master].[dbo].[xp_readerrorlog]
        @LogNum, 1, NULL, NULL, @StartDate, @EndDate;
  END;
  IF @ExcludeLogonSurroundingEvents = 1
    DELETE FROM #Logs WHERE ProcessInfo = 'Logon';
  SELECT * FROM #Logs ORDER BY LogDate DESC;
END
```

I am not going to put the full list of possible errors here – it may be excessive and, in many cases, is system specific. But you need to analyze any suspicious data from the output and understand its possible impact on the system.

Finally, I suggest setting up alerts for high-severity errors in SQL Server Agent, if this has not already been done.

Consolidating Instances and Databases

You can't talk about SQL Server troubleshooting without discussing database and SQL Server instances consolidation. While consolidating often reduces hardware and licensing costs, it doesn't come for free; you need to analyze its possible negative impact on the current or future system performance.

There is no universal consolidation strategy that can be used with every project. You should analyze the amount of data, load, hardware configuration, and your business and security requirements when making this decision. However, as a general rule, avoid consolidating OLTP and Data

Warehouse/Reporting databases on the same server when they are working under a heavy load (or, if they are consolidated, consider splitting them). Data warehouse queries usually process large amounts of data, which leads to heavy I/O activity and flushes the content of the buffer pool. Taken together, this negatively affects the performance of other systems.

In addition, analyze your security requirements when consolidating databases. Some security features, such as Audit, affect the entire server and add performance overhead for all databases on the server. *Transparent Data Encryption (TDE)* is another example: even though it is a database-level feature, SQL Server encrypts tempdb when either of the databases on the server has TDE enabled. This leads to performance overhead for all other systems. As a general rule, do not keep databases with different security requirements on the same instance of SQL Server. Look at the trends and spikes in metrics and separate databases from each other when needed. (I will provide code to help you analyze CPU, I/O and Memory usage on a per-database basis later in the book.)

I suggest utilizing virtualization and consolidating multiple VMs on one or a few hosts, instead of putting multiple independent and active databases on a single SQL Server instance. This will give you much better flexibility, manageability, and isolation between the systems, especially if multiple SQL Server instances are running on the same server. It is much easier to manage their resource consumption when you virtualize them.

Observer Effect

The production deployment of every serious SQL Server system requires implementing a monitoring strategy. This may include third-party monitoring tools, code built based on standard SQL Server technologies, or both.

A good monitoring strategy is essential for SQL Server production support. It helps you to be more proactive and reduces incident detection and recovery times. Unfortunately, it does not come for free–every type of monitoring adds the overhead to the system. In some cases, this overhead may be negligible and acceptable; in others it may significantly affect server performance.

During my career as an SQL Server consultant, I've seen many cases of inefficient monitoring. For example, one client was using a tool that provided information about index fragmentation by calling the sys.dm_db_index_physical_stats function, in DETAILED mode, every four hours for every index in the database. This introduced huge spikes in I/O and cleared the buffer pool, leading to a noticeable performance hit. Another client used a tool that constantly polled various DMVs, adding significant CPU load to the server.

Fortunately, in many cases, you will be able to see those queries and evaluate their impact during system troubleshooting. This is not always the case, however, with other technologies, for example with monitoring based on Extended Events. (I will talk about methods for detecting inefficient queries in Chapter 4 and about Extended Events in Appendix A.) Extended Events is a great technology that allows you to troubleshoot complex problems in SQL Server. It is not, however, the best choice as a profiling tool. Some events are heavy and may introduce large overhead in busy environments.

Let's look at the example and create an xEvents session that captures queries running in the system, as shown in Listing 1-5.

Listing 1-5. Creating an xEvents session to capture queries in the system.

```
CREATE EVENT SESSION CaptureQueries ON SERVER
ADD EVENT sqlserver.rpc_completed
(
  SET collect_statement=(1)
  ACTION
  (
    sqlos.task_time,sqlserver.client_app_name
    ,sqlserver.client_hostname
    ,sqlserver.database_name
    ,sqlserver.nt_username
    ,sqlserver.sql_text
  )
),
ADD EVENT sqlserver.sql_batch_completed
(
  ACTION
  (
    sqlos.task_time
    ,sqlserver.client_app_name
```

```
    ,sqlserver.client_hostname
    ,sqlserver.database_name
    ,sqlserver.nt_username
    ,sqlserver.sql_text
  )
),
ADD EVENT sqlserver.sql_statement_completed
ADD TARGET package0.event_file
(SET FILENAME=N'C:\PerfLogs\LongSql.xel',MAX_FILE_SIZE=(200))
WITH
(
  MAX_MEMORY =4096 KB
  ,EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS
  ,MAX_DISPATCH_LATENCY=5 SECONDS
);
```

Next, deploy it to the server that operates under a heavy load with a large number of concurrent requests. Measure the throughput in the system, with and without xEvents session running. Obviously, be careful—and don't run it on the production server!

Figure 1-5 illustrates CPU load and number of batch requests per second in both scenarios on one of my servers. As you can see, enabling the xEvents session decreased throughput by more than 20%. To make matters worse, it would be very hard to detect the existence of that session on the server.

Figure 1-5. Server throughput with and without an active xEvents session.

Obviously, the degree of impact would depend on the system's workload. In either case, check for any unnecessary monitoring or data-collection tools when you do the troubleshooting.

The bottom line: Evaluate the monitoring strategy and estimate its overhead as part of your analysis, especially when the server hosts multiple databases. For example, Extended Events work at the server level. While you can filter the events based on database_id field, the filtering occurs after an event has been fired. This can affect all databases on the server.

Summary

System troubleshooting is a holistic process that requires you to analyze your entire application ecosystem. You need to look at hardware, OS and

virtualization layers, and at SQL Server and database configuration and adjust them as needed.

SQL Server provides many settings that you can use to fine-tune the installation to the system workload. There are also best practices that apply to most systems, including enabling IFI and Optimize for Ad-Hoc Workloads settings, increasing the number of files in tempdb, turning on some trace flags, disabling Auto Shrink, and setting up correct auto-growth parameters for database files.

In the next chapter, we'll talk about one of the most important components in SQL Server, SQLOS, and a troubleshooting technique called Wait Statistics.

Troubleshooting Checklist

- Perform a high-level analysis of hardware, network and disk subsystem
- Discuss host configuration and load in virtualized environments with infrastructure engineers
- Check OS and SQL Server versions, editions and patching level
- Check if *instant file initialization* is enabled
- Analyze trace flags
- Enable *Optimize for Ad-Hoc Workloads*
- Check memory and parallelism settings on the server
- Look at tempdb settings (including number of files); check for trace flag T1118 and potentially T1117, in SQL Server versions prior to 2016
- Disable *Auto Shrink* for databases
- Validate data and t-log file settings
- Check number of VLFs in transaction log files

- Check errors in SQL Server Log
- Check for unnecessary monitoring in the system

---

1 For a deep dive into data partitioning and disaster recovery strategies, please see my book *Pro SQL Server Internals (2 nd ed., Apress, 2016).*

Chapter 2. SQL Server Execution Model and Wait Statistics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 2 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at dmitri@aboutsqlserver.com.

It is impossible to troubleshoot SQL Server instances without understanding its execution model. You need to know how SQL Server runs tasks and manages resources if you want to detect bottlenecks in the system. We will cover those questions in this chapter.

First, the chapter will describe SQL Server's architecture and major components. Next, it will discuss SQL Server's execution model and introduce you to the popular troubleshooting technique called Wait Statistics. It will also cover several data management views commonly used during troubleshooting. Finally, it will provide you an overview of Resource Governor, which you can configure to segregate different workloads in the system.

SQL Server: High-Level Architecture

As you know, SQL Server is a very complex product that consists of dozens of components and subsystems. It is impossible to cover all of them here, but in this section, you'll get a high-level overview. For the sake of understanding, we'll divide these components and subsystems into seven categories, as shown in Figure 2-1. Let's talk about them.

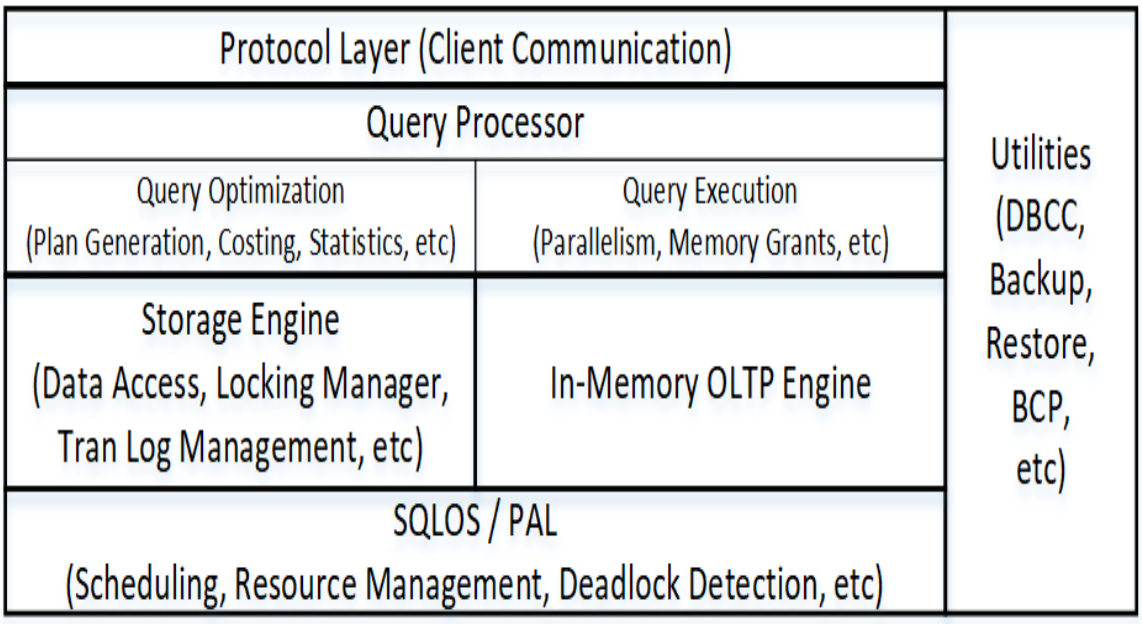

Figure 2-1. Major SQL Server Components

The *Protocol Layer* handles communication between SQL Server and client applications. It uses an internal format called *Tabular Data Stream (TDS)* to transmit data using network protocols such as TCP/IP or Name Pipes. If a client application and SQL Server are running on the same machine, you can use another protocol called *Shared Memory*.

> **NOTE**
>
> It is worth checking what protocols are enabled when you troubleshoot client connectivity issues. Some SQL Server editions, for example Express and Developer, disable TCP/IP and Name Pipes by default. They do not accept remote client connections until you enable network protocols in the *SQL Server Configuration Manager* utility.

There is also a large set of *utilities*, which are responsible for backup and restore, data import and export, executing DBCC commands, and many other actions.

The *Query Processor* layer is responsible for query optimization and execution. It parses, optimizes and manages compiled plans for the queries, and orchestrates all aspects of query execution.

The *Storage Engine* is responsible for data access and management in SQL Server. It works with the data on disk, manages transaction logs, and handles transactions, locking and concurrency along with a few other functions.

The *In-Memory OLTP Engine* supports In-Memory OLTP in SQL Server. It works with memory-optimized tables and is responsible for data management and access to those tables, native compilation, data persistence, and all other aspects of the technology.

There are layers of abstraction between the components. For example, *Query Interop* (not shown in Figure 2-1) allows the Query Processor to work with row-based and memory-optimized tables, transparently routing requests either to Storage or to In-Memory OLTP engines.

The most critical abstraction layer is *SQL Server Operating System (SQLOS)*, which isolates other SQL Server components from the operating systems and deals with scheduling, resource management and monitoring, exception handling, and many other aspects of SQL Server behavior. For example, when any SQL Server component needs to allocate memory, it does not call OS API functions: it requests memory from SQLOS. This allows SQL Server granular control over execution and internal resource usage without relying on the OS.

Finally, since the introduction of Linux support in SQL Server 2017, there is another component called *Platform Abstraction Layer (PAL)*, which serves as a proxy between SQLOS and operating systems. Except for few performance-critical use cases, SQLOS does not call OS API directly, relying on PAL instead. This architecture allows SQL Server's code to remain almost identical in Windows and Linux, which significantly speeds up development and product improvements.

From a troubleshooting standpoint, you'll see very little difference between SQL Server on Windows and on Linux. Obviously, you'll use different techniques when analyzing the SQL Server ecosystem and OS configuration. However, both platforms behave the same when you start to analyze issues *inside* SQL Server, so I am not going to differentiate between them in this book.

Let's look at the layers in more detail, beginning with SQLOS.

SQLOS and the Execution Model

Database servers are expected to handle a large number of user requests, and SQL Server is no exception. On a very high level, SQL Server assigns those requests to separate threads, executing requests simultaneously. Except in cases when the server is idle, the number of active threads exceeds the number of CPUs in the system, and efficient scheduling is the key to good server performance.

Early versions of SQL Server relied on Windows scheduling. Unfortunately, Windows (and Linux) are general purpose OSs, which means they use *preemptive scheduling*. They allocate a time interval, or *time quantum*, to a thread to run, then switch to other threads when it expires. This is an expensive operation that requires switching between user and kernel modes, negatively affecting system performance.

In SQL Server 7.0, Microsoft introduced the first version of *User Mode Scheduler (UMS)*-a thin layer between Windows and SQL Server that was primarily responsible for scheduling. It used *cooperative scheduling*, with SQL Server threads coded to voluntarily yield every 4ms, allowing other threads to execute. This approach significantly reduced expensive context switching in the system.

> **NOTE**
>
> Some SQL Server processes, like extended stored procedures, CLR routines, external languages and a few others, may still run in preemptive scheduling mode.

Microsoft continued to make improvements in UMS in SQL Server 2000 and, finally, in SQL Server 2005 redesigned it to the much more robust SQLOS. In later versions of SQL Server, SQLOS is responsible for scheduling, memory and I/O management, exception handling, CLR and external languages hosting, and quite a few other functions.

When you start an SQL Server process, SQLOS creates a set of *schedulers* that manage workload across CPUs. The number of schedulers matches the number of logical CPUs in the system, with an additional scheduler created for a *Dedicated Admin Connection (DAC)*. For example, if you have two quad-core physical CPUs with hyper-threading enabled, SQL Server will create 17 schedulers in the system. For all practical purposes, you can think of schedulers as the CPUs; I will use those terms interchangeably throughout the book.

> **NOTE**
>
> The Dedicated Admin Connection is your *last resort* troubleshooting connection. It allows you to access SQL Server if it becomes unresponsive and does not accept normal connections. I will talk about it in Chapter 13.

Each scheduler will be in an `ONLINE` or `OFFLINE` state, depending on its affinity mask setting and core-based licensing model. The schedulers usually do not migrate between CPUs; however, it is possible, especially under heavy load. Nevertheless, in most cases this behavior does not affect the troubleshooting process.

The schedulers are responsible to manage the set of *worker threads*, sometimes called *workers*. The maximum number of workers in a system is specified by the *Max Worker Thread* configuration option. The default value of zero indicates that SQL Server calculates the maximum number of worker threads based on number of schedulers in the system. In most cases, you do not need to change this default value—in fact, don't change it unless you know *exactly* what you are doing.

Each time there is a task to execute, it is assigned to an idle worker. When there are no idle workers, the scheduler creates a new one. It also destroys idle workers after 15 minutes of inactivity or in case of memory pressure. Each worker uses 512KB of RAM in 32-bit and 2MB of RAM in 64-bit SQL Server for the thread stack.

Workers do not move between schedulers; tasks do not move between workers. SQLOS, however, can create child tasks and assign them to different workers, for example in the case of parallel execution plans. This may explain situations when some schedulers are running under heavier loads than others – some workers could end up with more expensive tasks from time to time. You can think about *workers* as the logical representation of OS threads, and *tasks* as the unit of works those threads handle.

In most cases, we focus on tasks during troubleshooting. There is an exception, however: when a task is in the `PENDING` state, which means that the task has been created and is waiting for available workers . This is completely normal, and workers are usually assigned to tasks very quickly. However, it can also indicate a very dangerous condition when the system does not have enough workers to handle the requests. I will discuss how to detect and address that issue in Chapter 13.

Besides `PENDING`, a task may be in five other possible states:

`RUNNING`

The task is currently executing on the scheduler.

`RUNNABLE`

The task is waiting for the scheduler to be executed.

`SUSPENDED`

The task is waiting for an external event or resource.

`SPINLOOP`

The task is processing a *spinlock*. *Spinlocks* are synchronization objects that protect some internal objects. SQL Server may use them when it expects that access to the object will be granted very quickly, avoiding context switching for the workers. I will talk about troubleshooting spinlock issues in Chapter 13.

DONE

The task is complete.

The first three states are the most important and common. Each scheduler has at most one task in the RUNNING state. In addition, it has two different queues—one for RUNNABLE and one for SUSPENDED tasks. When the RUNNING task needs some resources—a data page from a disk, for example—it submits an I/O request and changes the state to SUSPENDED. It stays in the SUSPENDED queue until the request is fulfilled and the page has been read. After that, when it is ready to resume execution, the task is moved to the RUNNABLE queue.

Perhaps the closest real-life analogy to this process is a grocery-store checkout line. Think of cashiers as schedulers and customers as tasks in the RUNNABLE queue. A customer who is currently checking out is similar to a task in the RUNNING state.

If item is missing a UPC code, a cashier sends a store worker to do a price check. The cashier suspends the checkout process for the current customer, asking her or him to step aside (to the SUSPENDED queue). When the worker comes back with the price information, the customer moves to the end of the checkout line (the end of the RUNNABLE queue).

Of course, SQL Server's execution is much more efficient than a real-life store, where customers must wait patiently in line for the price check to complete. (A customer who is forced to move to the end of the RUNNABLE queue would probably wish for such efficiency!)

Wait Statistics

With exception of initialization and clean-up, a task spends its time switching between RUNNING, SUSPENDED, and RUNNABLE states, as shown in Figure 2-2. The total execution time will include time in RUNNING state, when task actually executed; time in RUNNABLE state, when the task is waiting for scheduler (CPU) to execute; and time in SUSPENDED state, when task is waiting for resources.

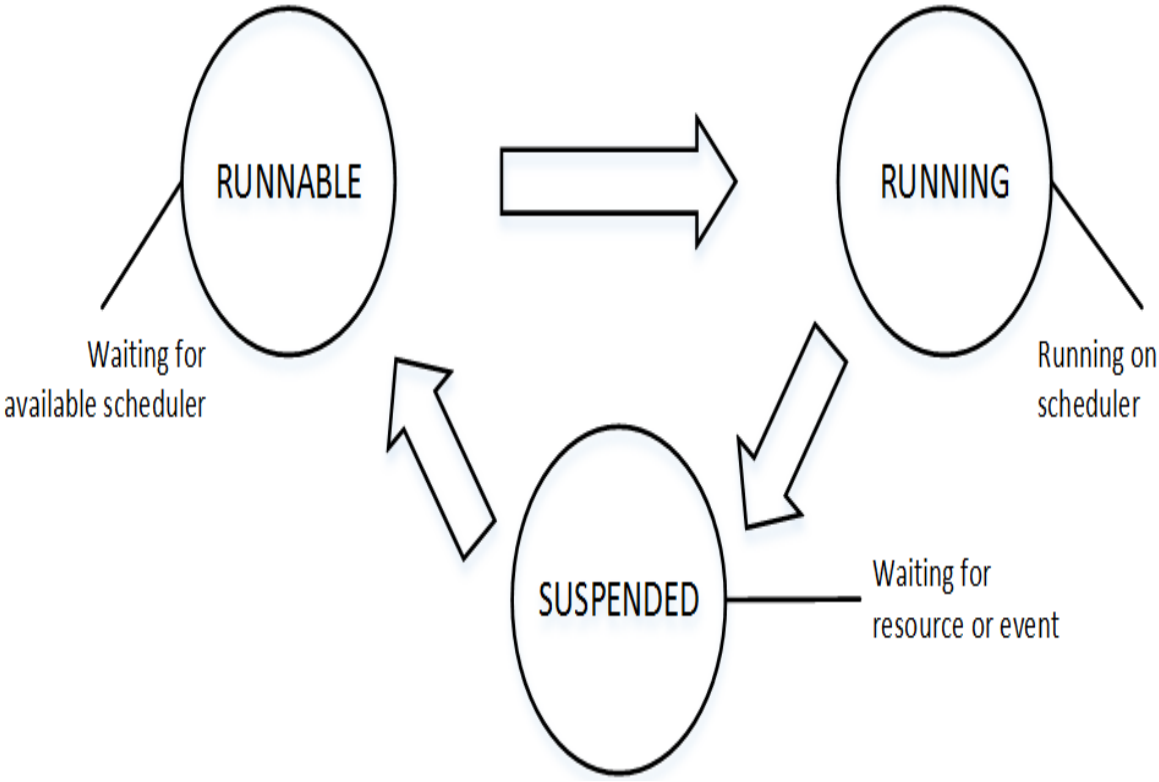

Figure 2-2. Task life cycle

In a nutshell, the goal of any performance-tuning process is improving system throughput by reducing query execution times. You can achieve this by reducing the time that query tasks spend in any of those states.

You can decrease query RUNNING time by upgrading hardware and moving to faster CPUs or by reducing amount of work tasks perform with query optimization.

You can shrink RUNNABLE time by adding more CPU resources or reducing the load on the system.

However, in most cases, you will get the most benefit by focusing on the time that tasks spend in SUSPENDED state while waiting for resources.

SQL Server tracks the cumulative time tasks spend in SUSPENDED state for different types of waits. You can view this data through the sys.dm_os_wait_stats view to get a quick sense of the main bottlenecks in your system and further fine-tune your troubleshooting strategy.

The code in Listing 2-1 shows you the wait types that take the most time in your system (filtering out some nonessential wait types, mainly related to internal SQL Server processes). The data is collected from the time of the last SQL Server restart, or since you last cleared it with the DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR) command. Each new SQL Server version introduces new wait types. Some are useful for troubleshooting; others are benign and will need to be filtered out.[1]

Example 2-1. Getting top wait types in the system

```
;WITH Waits
AS
(
  SELECT
    wait_type, wait_time_ms, waiting_tasks_count,signal_wait_time_ms
    ,wait_time_ms - signal_wait_time_ms AS resource_wait_time_ms
    ,100. * wait_time_ms / SUM(wait_time_ms) OVER() AS Pct
    ,ROW_NUMBER() OVER(ORDER BY wait_time_ms DESC) AS RowNum
  FROM sys.dm_os_wait_stats WITH (NOLOCK)
  WHERE
    wait_type NOT IN /* Filtering out non-essential system waits */

(N'BROKER_EVENTHANDLER',N'BROKER_RECEIVE_WAITFOR',N'BROKER_TASK_STOP
'

,N'BROKER_TO_FLUSH',N'BROKER_TRANSMITTER',N'CHECKPOINT_QUEUE',N'CHKP
T'
    ,N'CLR_SEMAPHORE',N'CLR_AUTO_EVENT',N'CLR_MANUAL_EVENT'

,N'DBMIRROR_DBM_EVENT',N'DBMIRROR_EVENTS_QUEUE',N'DBMIRROR_WORKER_QU
EUE'

,N'DBMIRRORING_CMD',N'DIRTY_PAGE_POLL',N'DISPATCHER_QUEUE_SEMAPHORE'

,N'EXECSYNC',N'FSAGENT',N'FT_IFTS_SCHEDULER_IDLE_WAIT',N'FT_IFTSHC_M
UTEX'
```

```
        ,N'HADR_CLUSAPI_CALL',N'HADR_FILESTREAM_IOMGR_IOCOMPLETION'
    ,N'HADR_LOGCAPTURE_WAIT',N'HADR_NOTIFICATION_DEQUEUE'

,N'HADR_TIMER_TASK',N'HADR_WORK_QUEUE',N'KSOURCE_WAKEUP',N'LAZYWRITE
R_SLEEP'
    ,N'LOGMGR_QUEUE',N'MEMORY_ALLOCATION_EXT',N'ONDEMAND_TASK_QUEUE'
    ,N'PARALLEL_REDO_WORKER_WAIT_WORK',N'PARALLEL_REDO_DRAIN_WORKER'
    ,N'PARALLEL_REDO_LOG_CACHE',N'PARALLEL_REDO_TRAN_LIST'
    ,N'PARALLEL_REDO_WORKER_SYNC',N'PREEMPTIVE_HADR_LEASE_MECHANISM'
    ,N'PREEMPTIVE_SP_SERVER_DIAGNOSTICS',N'PREEMPTIVE_OS_LIBRARYOPS'

,N'PREEMPTIVE_OS_COMOPS',N'PREEMPTIVE_OS_CRYPTOPS',N'PREEMPTIVE_OS_P
IPEOPS'
    ,N'PREEMPTIVE_OS_AUTHENTICATIONOPS',N'PREEMPTIVE_OS_GENERICOPS'
    ,N'PREEMPTIVE_OS_VERIFYTRUST',N'PREEMPTIVE_OS_FILEOPS'
    ,N'PREEMPTIVE_OS_DEVICEOPS',N'PREEMPTIVE_OS_QUERYREGISTRY'
    ,N'PREEMPTIVE_OS_WRITEFILE',N'PREEMPTIVE_XE_CALLBACKEXECUTE'
    ,N'PREEMPTIVE_XE_DISPATCHER',N'PREEMPTIVE_XE_GETTARGETSTATE'
    ,N'PREEMPTIVE_XE_SESSIONCOMMIT',N'PREEMPTIVE_XE_TARGETINIT'

,N'PREEMPTIVE_XE_TARGETFINALIZE',N'PWAIT_ALL_COMPONENTS_INITIALIZED'

,N'PWAIT_DIRECTLOGCONSUMER_GETNEXT',N'PWAIT_EXTENSIBILITY_CLEANUP_TA
SK'
    ,N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP',N'QDS_ASYNC_QUEUE'
    ,N'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP'

,N'REQUEST_FOR_DEADLOCK_SEARCH',N'RESOURCE_QUEUE',N'SERVER_IDLE_CHEC
K'
    ,N'SLEEP_BPOOL_FLUSH',N'SLEEP_DBSTARTUP',N'SLEEP_DCOMSTARTUP'

,N'SLEEP_MASTERDBREADY',N'SLEEP_MASTERMDREADY',N'SLEEP_MASTERUPGRADE
D'
    ,N'SLEEP_MSDBSTARTUP',N'SLEEP_SYSTEMTASK',N'SLEEP_TASK'

,N'SLEEP_TEMPDBSTARTUP',N'SNI_HTTP_ACCEPT',N'SOS_WORK_DISPATCHER'
    ,N'SP_SERVER_DIAGNOSTICS_SLEEP',N'SQLTRACE_BUFFER_FLUSH'
    ,N'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',N'SQLTRACE_WAIT_ENTRIES'
    ,N'STARTUP_DEPENDENCY_MANAGER',N'WAIT_FOR_RESULTS'
    ,N'WAITFOR',N'WAITFOR_TASKSHUTDOWN',N'WAIT_XTP_HOST_WAIT'

,N'WAIT_XTP_OFFLINE_CKPT_NEW_LOG',N'WAIT_XTP_CKPT_CLOSE',N'WAIT_XTP_
RECOVERY'

,N'XE_BUFFERMGR_ALLPROCESSED_EVENT',N'XE_DISPATCHER_JOIN',N'XE_DISPA
TCHER_WAIT'
    ,N'XE_LIVE_TARGET_TVF',N'XE_TIMER_EVENT')
)
```

```
SELECT
  w1.wait_type AS [Wait Type]
  ,w1.waiting_tasks_count AS [Wait Count]
  ,CONVERT(DECIMAL(12,3), w1.wait_time_ms / 1000.0) AS [Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.wait_time_ms / w1.waiting_tasks_count)
        AS [Avg Wait Time]
  ,CONVERT(DECIMAL(12,3), w1.signal_wait_time_ms / 1000.0)
        AS [Signal Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.signal_wait_time_ms /
w1.waiting_tasks_count)
        AS [Avg Signal Wait Time]
  ,CONVERT(DECIMAL(12,3), w1.resource_wait_time_ms / 1000.0)
        AS [Resource Wait Time]
  ,CONVERT(DECIMAL(12,1), w1.resource_wait_time_ms /
w1.waiting_tasks_count)
        AS [Avg Resource Wait Time]
  ,CONVERT(DECIMAL(6,3), w1.Pct)
        AS [Percent]
  ,CONVERT(DECIMAL(6,3), w1.Pct + ISNULL(w2.Pct,0))
        AS [Running Percent]
FROM
        Waits w1 CROSS APPLY
        (
                SELECT SUM(w2.Pct) AS Pct
                FROM Waits w2
                WHERE w2.RowNum < w1.RowNum
        ) w2
WHERE
        w1.RowNum = 1 OR w2.Pct <= 99
ORDER BY
        w1.RowNum
OPTION (RECOMPILE, MAXDOP 1);
```

Figure 2-3 shows the output of this code from one of the production servers, early in the troubleshooting process. I can immediately see that majority of the waits in the system relate to blocking (LCK*) and I/O (PAGEIOLATCH*). This makes it much easier to decide where to focus my troubleshooting efforts.

	Wait Type	Wait Count	Wait Time	Avg Wait Time	Signal Wait Time	Avg Signal Wait Time	Resource Wait Time	Avg Resource Wait Time	Percent	Running Percent
1	LCK_M_U	538312358	2952904.553	5.0	278904.170	1.0	2674000.376	4.0	58.316	58.316
2	PAGEIOLATCH_SH	132056495	730022.059	5.0	17938.737	0.0	712083.322	5.0	14.417	72.733
3	LCK_M_S	196405075	379378.938	1.0	24706.314	0.0	354672.624	1.0	7.492	80.226
4	ASYNC_NETWORK_IO	36665258	254793.758	6.0	100063.339	2.0	154730.419	4.0	5.032	85.258
5	LOGBUFFER	11718571	165042.270	14.0	18931.562	1.0	146110.708	12.0	3.259	88.517
6	PAGEIOLATCH_EX	153474407	133057.566	0.0	3225.941	0.0	129831.625	0.0	2.628	91.145
7	LCK_M_IX	496185	98525.504	198.0	139.082	0.0	98386.422	198.0	1.946	93.091
8	IO_COMPLETION	93217317	81833.420	0.0	3505.294	0.0	78328.126	0.0	1.616	94.707
9	LATCH_EX	49863173	65876.146	1.0	10921.396	0.0	54954.750	1.0	1.301	96.008
10	ASYNC_IO_COMPLETION	57845	56036.933	968.0	22.078	0.0	56014.855	968.0	1.107	97.114
11	LCK_M_IS	57448	31694.644	551.0	9.403	0.0	31685.241	551.0	0.626	97.740
12	LCK_M_SCH_M	2228	31016.126	13921.0	0.918	0.0	31015.208	13920.0	0.613	98.353
13	WRITELOG	1969821	26014.687	13.0	715.277	0.0	25299.410	12.0	0.514	98.867
14	OLEDB	3041936	14911.992	4.0	6058.799	1.0	8853.193	2.0	0.294	99.161

Figure 2-3. Example of sys.dm_os_wait_stats output

This troubleshooting approach is called *Wait Statistics Analysis*. It's the one of the most frequently used troubleshooting and performance-tuning techniques in SQL Server. Figure 2-4 illustrates a typical troubleshooting cycle using Wait Statistics Analysis.

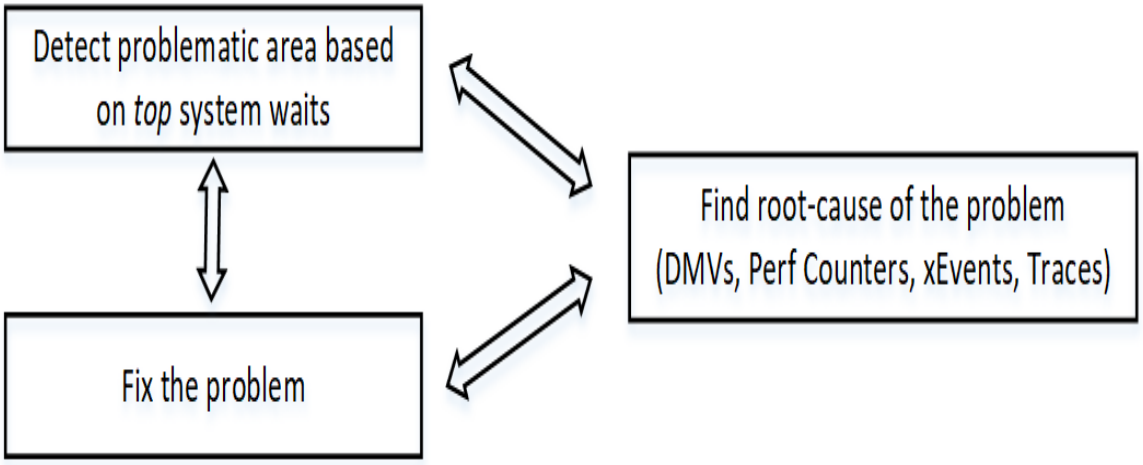

Figure 2-4. Typical Wait Statistics Analysis Troubleshooting Cycle

First, you identify the main bottleneck in the system by analyzing the top waits. Next, you confirm it with other tools and techniques and pinpoint the root cause of the problem. Finally, you fix it and repeat the cycle.

> **NOTE**
>
> A word of caution: This process may never end. While there are always opportunities to make things better, at some point further improvements become impractical. Remember the Pareto Principle – you will get 80% of improvements by spending 20% of your time – and don't waste time on nonessential tuning.

This looks very easy in theory; unfortunately, it is more complicated in real life. Many issues are related to each other, which can hide the real causes of bottlenecks. To choose a very common example: excessive disk waits are often triggered not by bad I/O performance, but by poorly optimized queries that constantly flush the buffer pool and overload the disk subsystem.

Figure 2-5 shows some of the high-level dependencies you might run into. This diagram is by no means exhaustive, but it illustrates the danger of tunnel vision during troubleshooting.

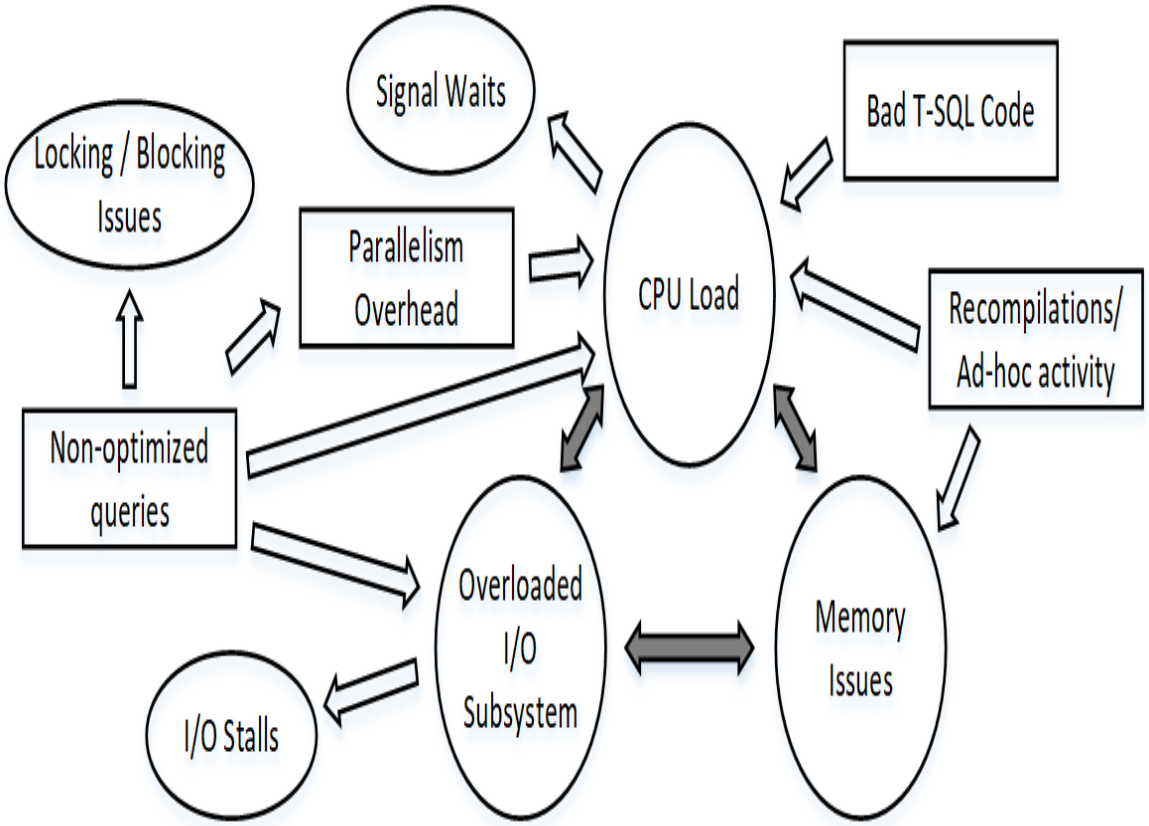

Figure 2-5. Dependencies and Issues

I considered listing the most common waits and possible root causes here, but I don't want you to start chasing symptoms rather than causes. Rather than jumping right to a list, read the book first so that you can understand the possible dependencies involved.

I'll start going through specific issues and troubleshooting techniques in the upcoming chapters, but for now, let's cover important data management views in SQL Server related to SQLOS and the SQL Server execution model.

Execution Model–Related Data Management Views

SQL Server comes with a very large number of data management views (DMVs). For details on all of them, you can consult the Microsoft

Documentation. Here, I will focus on just a small subset that I regularly use during troubleshooting. We will look at many others throughout the book.

sys.dm_os_wait_stats

As you saw earlier, the `sys.dm_os_wait_stats` view provides information about waits in the system. It will tell you how many times the wait occurs (`waiting_task_count`) along with cumulative times for resource (`resource_wait_time_ms`) and signal (`signal_wait_time_ms`) waits. The resource wait time indicates how long a task waited for the resource staying in SUSPENDED queue. The signal wait indicates the wait for the CPU in RUNNABLE queue after the resource wait was over.

For example, let's say a task is requested to read a data page from disk. The I/O request might take 6ms; then, the task might wait for another millisecond to resume the execution. If you view the wait data for this, you'll see 6ms of resource waits, 1ms of signal waits, and 7ms of total wait time.

Listing 2-2 shows you how to compare cumulative signal and resource waits in the system.

Example 2-2. Signal vs. resource waits

```
SELECT
    SUM(signal_wait_time_ms) AS [Signal Wait Time (ms)]
    ,CONVERT(DECIMAL(7,4), 100.0 * SUM (signal_wait_time_ms) /
        SUM(wait_time_ms)) AS [% Signal waits]
    ,SUM(wait_time_ms - signal_wait_time_ms) AS [Resource Wait Time
(ms)]
    ,CONVERT (DECIMAL(7,4), 100.0 * sum(wait_time_ms -
signal_wait_time_ms) /
        SUM(wait_time_ms)) AS [% Resource waits]
FROM
    sys.dm_os_wait_stats WITH (NOLOCK);
```

In most cases, signal waits should not exceed 10 to 15% of total wait time. A higher number may indicate a CPU bottleneck, with tasks spending a lot of time in the RUNNABLE queue. Do not jump to the conclusion that you need to

add more CPUs, though—it may be entirely possible to address the problem with performance tuning.

Pay attention to how often waits occur. Sometimes, you'll see waits with a low `waiting_task_count` and high total wait time. Depending on the situation, you may or may not want to analyze them, especially during the initial phase of troubleshooting. Such waits are often triggered by production incidents.

Finally, make sure that you are working with representative data. As I mentioned, statistics are collected from the time of the last SQL Server restart, and workload on the server may change over time.

I usually ask customers to clear the waits a few days before starting the troubleshooting. It is safe to use the `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` command in production, although it may affect data collection in some third-party monitoring tools. As another option, you can collect two separate snapshots of wait statistics and calculate the delta between them. I am including the script to do that to the companion materials of the book.

sys.dm_exec_session_wait_stats

Starting with SQL Server 2016, you can look at waits on the session level, using the `sys.dm_exec_session_wait_stats` view. This is extremely useful when you need to troubleshoot performance of long-running queries or slow stored procedures in the system. The view will show you the waits that occurred during execution and help you pinpoint bottlenecks and areas to research.

The columns and data in this view are similar to those in `sys.dm_os_wait_stats`; you can easily adjust scripts to work in both scenarios. Remember that data in `sys.dm_exec_session_wait_stats` clears when a session opens and when the pooled connection resets.

You may notice that the data is not always updated for currently running statements. You need to wait until a query completes for the data to become available.

sys.dm_os_waiting_tasks

The `sys.dm_os_waiting_tasks` view shows you a list of tasks that are *currently* waiting in the `SUSPENDED` queue. This view is handy when the server is overloaded or unresponsive and you want to understand why sessions were suspended. It is also very helpful when you troubleshoot concurrency issues and active blocking in the system, because it shows you the session ID of the blocker for the task (more in Chapter 8).

The most useful columns in this view are:

`session_id`

ID of the waiting session.

`wait_type`

Type of wait the session is waiting for.

`wait_duration_ms`

The duration of the wait.

`blocking_session_id`

The session blocking the current task. As I mention, this column is extremely useful when you troubleshoot active blocking in the system.

`resource_address`

Information on the resource the task is waiting for.

You may have more than one row per session in the output when you deal with parallel execution plans.

sys.dm_exec_requests

The `sys.dm_exec_requests` view provides detailed information on each request that is executing on the server. This gives you a great at-a-glance snapshot of what is happening now and allows you to pinpoint most CPU or I/O intensive queries *currently* running in the system.

This view will return information for both user and system sessions. You can filter out most system sessions by using `WHERE session_id > 50` predicate, although you may have some system sessions with id greater than 50 nowadays.

The most useful columns in this view are:

`session_id`

: The ID for the session. Unlike with `sys.dm_os_waiting_tasks`, you get a single row in the output per session.

`start_time`

: The time when the request started.

`total_elapsed_time`

: The request's duration.

`status`

: The current request status (`RUNNING`, `RUNNABLE`, `SUSPENDED`, `SLEEPING`). `SLEEPING` status indicates an idle connection.

`wait_type`, *`wait_time`*, *`wait_resource`*, *`blocking_session_id`*

: These appear if the request is currently suspended. Like `sys.dm_os_waiting_tasks`, the `blocking_session_id` column is very useful when you are troubleshooting active blocking in the system.

cpu_time, *logical_reads*, *reads*, *writes*, *granted_query_memory*, *dop*

These provide you with execution metrics.

sql_handle, plan_handle

These allow you to obtain the statement and its execution plan.

Listing 2-3 shows you the code that returns information about currently running CPU-intensive requests, along with connection information.

Example 2-3. Using sys.dm_exec_requests view

```
SELECT
        er.session_id
        ,er.request_id
        ,DB_NAME(er.database_id) as [database]
        ,er.start_time
        ,CONVERT(DECIMAL(21,3),er.total_elapsed_time / 1000.) AS
[duration]
        ,er.cpu_time
        ,SUBSTRING(
                qt.text,
                (er.statement_start_offset / 2) + 1,
                ((CASE er.statement_end_offset
                        WHEN -1 THEN DATALENGTH(qt.text)
                        ELSE er.statement_end_offset
                END - er.statement_start_offset) / 2) + 1
        ) AS [statement]
        ,er.status
        ,er.wait_type
        ,er.wait_time
        ,er.wait_resource
        ,er.blocking_session_id
        ,er.last_wait_type
        ,er.reads
        ,er.logical_reads
        ,er.writes
        ,er.granted_query_memory
        ,er.dop
        ,er.row_count
        ,er.percent_complete
        ,es.login_time
        ,es.original_login_name
```

```
        ,es.host_name
        ,es.program_name
        ,c.client_net_address
        ,ib.event_info AS [buffer]
        ,qt.text AS [sql]
        ,p.query_plan
FROM
        sys.dm_exec_requests er WITH (NOLOCK)
                OUTER APPLY sys.dm_exec_input_buffer(er.session_id,
er.request_id) ib
                OUTER APPLY sys.dm_exec_sql_text(er.sql_handle) qt
                OUTER APPLY sys.dm_exec_query_plan(er.plan_handle) p
                LEFT JOIN sys.dm_exec_connections c WITH (NOLOCK) ON
                        er.session_id = c.session_id
                LEFT JOIN sys.dm_exec_sessions es WITH (NOLOCK) ON
                        er.session_id = es.session_id
WHERE
        er.status <> 'background'
        AND er.session_id > 50
ORDER BY
        er.cpu_time desc
OPTION (RECOMPILE, MAXDOP 1);
```

A word of caution: Setting a query execution plan with the sys.dm_exec_query_plan function is expensive. Comment it out if your server is running under heavy CPU load.

sys.dm_os_schedulers

I do not use the sys.dm_os_schedulers view very often, only from time to time. As you can guess by the name, this view provides information about schedulers in the system. You can use it to get information about schedulers' distribution across NUMA nodes and to analyze metrics from individual schedulers.

I've already shown you the code for the first use case in Chapter 1, but it is worth repeating. Check the count of schedulers in each NUMA node to see if the CPU affinity has been set correctly.

Example 2-4. NUMA nodes schedulers statistics

```
SELECT
        parent_node_id
        ,COUNT(*) as [Schedulers]
```

```
        ,SUM(current_tasks_count) as [Current]
        ,SUM(runnable_tasks_count) as [Runnable]
FROM sys.dm_os_schedulers
WHERE status = 'VISIBLE ONLINE'
GROUP BY parent_node_id;
```

The `current_tasks_count` and `runnable_tasks_count` columns provide the number of tasks in the `RUNNING` and `RUNNABLE` queues in each node. A large `runnable_tasks_count` number may indicate a CPU bottleneck. Remember, however, that the numbers show what is happening in the system *now* and may not be representative over time. It is better to see cumulative information, for example the percentage of signal waits (see Listing 2-2) or CPU load overtime (see Chapter 6).

There are many other columns in the view that provide scheduler-specific statistics, such as status, number of workers and tasks in various states, number of context switches, CPU consumption and a few others. Check the documentation for more details.

Resource Governor Overview

Resource Governor is an Enterprise Edition feature that allows you to segregate and throttle different workloads on the server. Although it's been available for quite a long time, I consider Resource Governor a niche feature – I rarely see it in the field. (You may even consider skipping this section, and coming back if and when you have to deal with it.) Although you don't need to set it up, remember to check if Resource Governor is configured in the system you are troubleshooting – incorrect configuration can seriously impact server throughput.

When enabled, Resource Governor separates the sessions between different *workload groups* by calling *classifier function* at the time of the session's login. The classifier function is a simple user-defined function where you can use various connection properties (login name, application name, client IP address, etc.) to choose between workload groups.

Each workload group has several parameters, such as `MAXDOP`, maximum allowed CPU time for the request, and maximum number of simultaneous requests allowed in the group. The workgroups are also associated with a *resource pool*, where you can customize resource usage for associated workload groups.

The SQL Server documentation refers to *resource pools* as "the virtual SQL Server instances inside of a SQL Server instance." I do not think this is an accurate definition, though, because resource pools do not provide enough isolation from each other. However, you can control and limit CPU bandwidth and affinity, along with query memory grants (see Chapter 7).

Starting with SQL Server 2014, you can also control disk throughput by limiting resource pool IOPS. **You cannot, however, control buffer pool usage – it is shared across all pools.**

There are two system workload groups and resource pools: *internal* and *default*. As you can guess by the names, the first handles internal workload. The second is responsible for all non-classified workload. You can change the parameters of the *default* workload group without creating other user-defined workload groups and pools.

Figure 2-6 shows a Resource Governor configuration for an example scenario in which you want to separate OLTP and reporting workloads. This will reduce the impact of reporting queries on critical OLTP transactions, preventing them from saturating CPU and I/O.

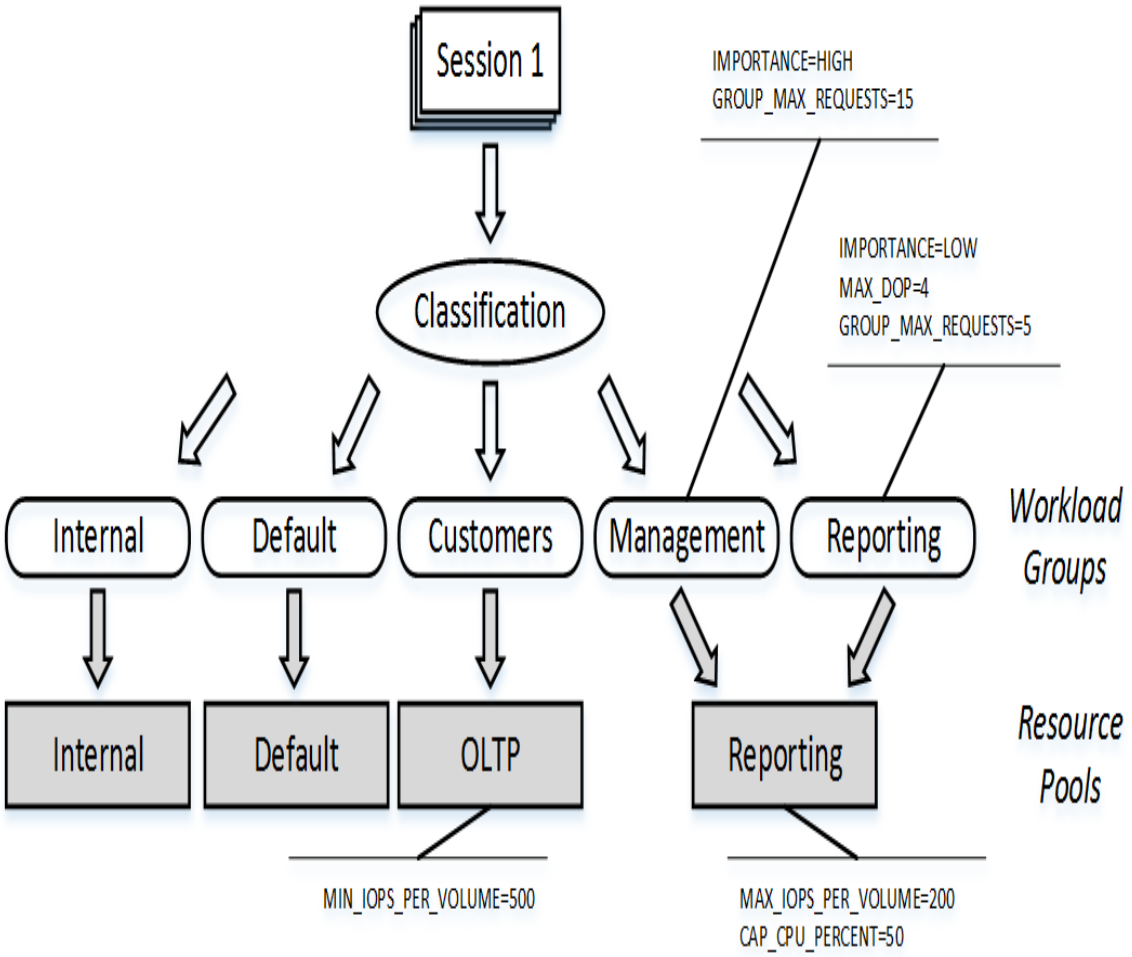

Figure 2-6. Example of Resource Governor Configuration

Resource Governor is useful, but it is not the easiest feature to configure and maintain. You need to do some planning and math when you want to configure resource throttling across multiple busy resource pools.

You also need to reevaluate the settings overtime, because hardware and workload requirements may change. I recently had to troubleshoot a case where a major disk subsystem upgrade did not improve system performance. We found that I/O in the system had been throttled by a `MAX_IOPS_PER_VOLUME` setting in the resource pool.

In conclusion, Resource Governor is good in use cases where you need to segregate different workloads in a single database on a standalone server or an instance that uses Failover Clustering. It is also useful for reducing the impact of database maintenance. For example, you can limit CPU utilized by

backup compression or I/O load from index maintenance by running them in a separate resource pool.

I recommend looking at different technologies when you need to segregate a different workload in the Always On Availability Groups setup. The readable secondaries may provide better scalability in the long term. In addition, when you need to segregate workloads from multiple databases running on a single SQL Server instance, it's usually better to split the databases across multiple instances, and potentially virtualize them.

Summary

SQLOS is the vital subsystem responsible for scheduling and resource management in SQL Server. At startup, it creates schedulers—one per logical CPU—allocating the pool of worker threads to each scheduler to manage. User and system tasks are assigned to the worker threads, which perform the actual work.

SQL Server uses cooperative scheduling, with workers voluntarily yielding every 4ms. The tasks constantly migrate through the `RUNNING`, `SUSPENDED`, and `RUNNABLE` states while they are running on CPU or waiting for CPU and resources. SQL Server tracks the different type of waits and provides that information in `sys.dm_os_wait_tasks` view. You can analyze the most common waits and identify bottlenecks in the system with the troubleshooting process called Wait Statistics.

Be careful when analyzing waits; don't jump to immediate conclusions. Many performance issues may be related and can mask each other. You'll need to identify and confirm the root cause of the problem as part of your analysis.

In the next chapter, we will dive deeper into troubleshooting particular issues, starting with the disk, and learn how to diagnose and address them.

Troubleshooting Checklist

- Look at the waits in the system. Make sure that wait statistics are representative.
- Analyze percentages of signal and resource waits.
- Validate Resource Governor configuration when present.
- Triage the waits, looking for bottlenecks.

---

1 The code in Listing 2-1 is good for versions up to SQL Server 2019. To exclude other wait types in future versions, see Microsoft's documentation.

About the Author

Dmitri Korotkevitch is a Microsoft Data Platform MVP and the Director of Database Services at Chewy.com. He specializes in the design, development, and performance tuning of complex OLTP systems that handle thousands of transactions per second, and has years of experience working with Microsoft SQL Server as an Application and Database Developer, Database Administrator, and Database Architect. Dmitri also provides SQL Server consulting services and training to clients around the world.