

Pro SQL Server 2019 Wait Statistics

A Practical Guide to Analyzing
Performance in SQL Server

—
Second Edition

—
Enrico van de Laar

Apress®

Pro SQL Server 2019 Wait Statistics

**A Practical Guide to Analyzing
Performance in SQL Server**

Second Edition

Enrico van de Laar

Apress®

Pro SQL Server 2019 Wait Statistics: A Practical Guide to Analyzing Performance in SQL Server

Enrico van de Laar
Drachten, The Netherlands

ISBN-13 (pbk): 978-1-4842-4915-4
<https://doi.org/10.1007/978-1-4842-4916-1>

ISBN-13 (electronic): 978-1-4842-4916-1

Copyright © 2019 by Enrico van de Laar

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaahr

Acquisitions Editor: Jonathan Gennick

Development Editor: Laura Berendson

Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York,
233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505,
e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California
LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc).
SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484249154. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*This book is dedicated to cats and pizza.
I had to leave both for a little while to write this book,
else it would probably still be a work in progress.*

Table of Contents

About the Author	xv
About the Technical Reviewers	xvii
Acknowledgments	xix
Introduction	xxi
Part I: Foundations of Wait Statistics Analysis.....	1
Chapter 1: Wait Statistics Internals.....	3
A Brief History of Wait Statistics	4
The SQLOS	6
Schedulers, Tasks, and Worker Threads.....	9
Sessions	10
Requests.....	11
Tasks	12
Worker Threads	13
Schedulers.....	16
Putting It All Together	17
Wait Statistics	19
Summary.....	23
Chapter 2: Querying SQL Server Wait Statistics	25
Sys.dm_os_wait_stats.....	26
Sys.dm_os_waiting_tasks.....	29
Understanding sys.dm_os_waiting_tasks	29
Querying sys.dm_os_waiting_tasks	31

TABLE OF CONTENTS

Sys.dm_exec_requests.....	33
Understanding sys.dm_exec_requests	33
Querying sys.dm_exec_requests	35
Sys.dm_exec_session_wait_stats.....	36
Combining DMVs to Detect Waits Right Now	38
Viewing Wait Statistics Using Perfmon	43
Capturing Wait Statistics Using Extended Events	45
Capture Wait Statistics Information for a Specific Query	46
Analyzing Wait Statistics on a Per-Query Basis Using Execution Plans	57
Summary.....	61
Chapter 3: The Query Store.....	63
What Is the Query Store?	63
Query Store Architecture.....	64
How Wait Statistics Are Processed in the Query Store	65
Accessing Wait Statistics Through the Query Store Reports.....	68
Accessing Wait Statistics Through the Query Store DMVs.....	70
Summary.....	73
Chapter 4: Building a Solid Baseline	75
What Are Baselines?	76
Visualizing Your Baselines.....	78
Baseline Types and Statistics	79
Baseline Pitfalls	81
Too Much Information.....	81
Know Your Metrics.....	81
Focus on the Big Measurement Changes.....	81
Use Fixed Intervals	82
Building a Baseline for Wait Statistics Analysis	82
Reset Capture Method.....	86
Delta Capture Method.....	87

TABLE OF CONTENTS

Using SQL Server Agent to Schedule Measurements.....	89
Wait Statistics Baseline Analysis	91
Summary.....	99
Part II: Wait Types.....	101
Chapter 5: CPU-Related Wait Types	103
CXPACKET	103
What Is the CXPACKET Wait Type?	104
Lowering CXPACKET Wait Time by Tuning the Parallelism Configuration	107
Lowering CXPACKET Wait Time by Resolving Skewed Workloads	111
Introduction of the CXCONSUMER Wait Type in SQL Server 2016 SP2 and 2017 CU3.....	112
CXPACKET Summary	113
SOS_SCHEDULER_YIELD.....	114
What Is the SOS_SCHEDULER_YIELD Wait Type?	114
Lowering SOS_SCHEDULER_YIELD Waits.....	117
SOS_SCHEDULER_YIELD Summary	122
THREADPOOL	123
What Is the THREADPOOL Wait Type?	123
THREADPOOL Example	126
Gaining Access to Our SQL Server During THREADPOOL Waits	130
Lowering THREADPOOL Waits Caused by Parallelism	132
Lowering THREADPOOL Waits Caused by User Connections	134
THREADPOOL Summary	137
Chapter 6: IO-Related Wait Types	139
ASYNC_IO_COMPLETION.....	139
What Is the ASYNC_IO_COMPLETION Wait Type?	140
ASYNC_IO_COMPLETION Example.....	141
Lowering ASYNC_IO_COMPLETION Waits.....	142
ASYNC_IO_COMPLETION Summary.....	147

TABLE OF CONTENTS

ASYNC_NETWORK_IO.....	147
What Is the ASYNC_NETWORK_IO Wait Type?	147
ASYNC_NETWORK_IO Example	148
Lowering ASYNC_NETWORK_IO Waits.....	149
ASYNC_NETWORK_IO Summary.....	151
CMEMTHREAD.....	151
What Is the CMEMTHREAD Wait Type?	151
Lowering CMEMTHREAD Waits.....	153
CMEMTHREAD Summary.....	154
IO_COMPLETION.....	154
What Is the IO_COMPLETION Wait Type?	155
IO_COMPLETION Example	155
Lowering IO_COMPLETION Waits.....	157
IO_COMPLETION Summary.....	157
LOGBUFFER and WRITELOG.....	157
What Are the LOGBUFFER and WRITELOG Wait Types?	158
LOGBUFFER and WRITELOG Example	160
Lowering LOGBUFFER and WRITELOG Waits.....	162
LOGBUFFER and WRITELOG Summary.....	163
RESOURCE_SEMAPHORE	163
What Is the RESOURCE_SEMAPHORE Wait Type?.....	163
RESOURCE_SEMAPHORE Example	165
Lowering RESOURCE_SEMAPHORE Waits	170
RESOURCE_SEMAPHORE Summary	171
RESOURCE_SEMAPHORE_QUERY_COMPILE.....	171
What Is the RESOURCE_SEMAPHORE_QUERY_COMPILE Wait Type?.....	172
RESOURCE_SEMAPHORE_QUERY_COMPILE Example.....	174
Lowering RESOURCE_SEMAPHORE_QUERY_COMPILE Waits	177
RESOURCE_SEMAPHORE_QUERY_COMPILE Summary	178

TABLE OF CONTENTS

SLEEP_BPOOL_FLUSH	179
What Is the SLEEP_BPOOL_FLUSH Wait Type?	179
SLEEP_BPOOL_FLUSH Example	182
Lowering SLEEP_BPOOL_FLUSH Waits	185
SLEEP_BPOOL_FLUSH Summary	185
WRITE_COMPLETION	186
What Is the WRITE_COMPLETION Wait Type?	186
WRITE_COMPLETION Example	186
Lowering WRITE_COMPLETION Waits	187
WRITE_COMPLETION Summary	187
Chapter 7: Backup-Related Wait Types.....	189
BACKUPBUFFER	190
What Is the BACKUPBUFFER Wait Type?	190
BACKUPBUFFER Example	193
Lowering BACKUPBUFFER Waits	194
BACKUPBUFFER Summary	195
BACKUPIO.....	195
What Is the BACKUPIO Wait Type?	196
BACKUPIO Example	196
Lowering BACKUPIO Waits	197
BACKUPIO Summary	198
BACKUPTHREAD	198
What Is the BACKUPTHREAD Wait Type?	198
BACKUPTHREAD Example	199
Lowering BACKUPTHREAD Waits	200
BACKUPTHREAD Summary	201
Chapter 8: Lock-Related Wait Types	203
Introduction to Locking and Blocking	205
Lock Modes and Compatibility	205
Locking Hierarchy	207
Isolation Levels	208

TABLE OF CONTENTS

Querying Lock Information	212
LCK_M_S	216
What Is the LCK_M_S Wait Type?	216
LCK_M_S Example	217
Lowering LCK_M_S Waits.....	218
LCK_M_S Summary.....	219
LCK_M_U	220
What Is the LCK_M_U Wait Type?.....	220
LCK_M_U Example	222
Lowering LCK_M_U Waits	223
LCK_M_U Summary	223
LCK_M_X.....	223
What Is the LCK_M_X Wait Type?	224
LCK_M_X Example	224
Lowering LCK_M_X Waits.....	225
LCK_M_X Summary.....	226
LCK_M_I[xx].....	226
What Is the LCK_M_I[xx] Wait Type?	226
LCK_M_I[xx] Example.....	227
Lowering LCK_M_I[xx] Waits	229
LCK_M_I[xx] Summary.....	229
LCK_M_SCH_S and LCK_M_SCH_M.....	230
What Are the LCK_M_SCH_S and LCK_M_SCH_M Wait Types?	230
LCK_M_SCH_S and LCK_M_SCH_M Example.....	231
Lowering LCK_M_SCH_S and LCK_M_SCH_M Waits.....	233
LCK_M_SCH_S and LCK_M_SCH_M Summary.....	233
Chapter 9: Latch-Related Wait Types.....	235
Introduction to Latches	235
Latch Modes	237
Latch Waits	238

TABLE OF CONTENTS

Sys.dm_os_latch_stats	240
Page-Latch Contention	241
PAGELATCH_[xx]	247
What Is the PAGELATCH_[xx] Wait Type?	247
PAGELATCH_[xx] Example	248
Lowering PAGELATCH_[xx] Waits.....	252
PAGELATCH_[xx] Summary.....	258
LATCH_[xx].....	258
What Is the LATCH_[xx] Wait Type?	259
LATCH_[xx] Example.....	259
Lowering LATCH_[xx] Waits.....	265
LATCH_[xx] Summary.....	266
PAGEIOLATCH_[xx]	266
What Is the PAGEIOLATCH_[xx] Wait Type?	267
PAGEIOLATCH_[xx] Example	269
Lowering PAGEIOLATCH_[xx] Waits	270
PAGEIOLATCH_[xx] Summary	277
Chapter 10: High-Availability and Disaster-Recovery Wait Types	279
DBMIRROR_SEND	280
What Is the DBMIRROR_SEND Wait Type?	283
DBMIRROR_SEND Example	283
Lowering DBMIRROR_SEND Waits.....	285
DBMIRROR_SEND Summary.....	286
HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE.....	287
What Are the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE Wait Types?	287
HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE Summary.....	290
HADR_SYNC_COMMIT.....	290
What Is the HADR_SYNC_COMMIT Wait Type?	290
HADR_SYNC_COMMIT Example.....	291
Lowering HADR_SYNC_COMMIT Waits.....	294
HADR_SYNC_COMMIT Summary.....	297

TABLE OF CONTENTS

REDO_THREAD_PENDING_WORK.....	297
What Is the REDO_THREAD_PENDING_WORK Wait Type?	298
REDO_THREAD_PENDING_WORK Summary.....	300
Chapter 11: Preemptive Wait Types.....	301
SQL Server on Linux.....	302
PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE	305
What Are the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Wait Types?	305
PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Example.....	306
Lowering PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Waits	312
PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Summary.....	313
PREEMPTIVE_OS_WRITEFILEGATHER	313
What Is the PREEMPTIVE_OS_WRITEFILEGATHER Wait Type?	313
PREEMPTIVE_OS_WRITEFILEGATHER Example	314
Lowering PREEMPTIVE_OS_WRITEFILEGATHER Waits.....	315
PREEMPTIVE_OS_WRITEFILEGATHER Summary.....	316
PREEMPTIVE_OS_AUTHENTICATIONOPS.....	316
What Is the PREEMPTIVE_OS_AUTHENTICATIONOPS Wait Type?	317
PREEMPTIVE_OS_AUTHENTICATIONOPS Example.....	318
Lowering PREEMPTIVE_OS_AUTHENTICATIONOPS Waits	319
PREEMPTIVE_OS_AUTHENTICATIONOPS Summary	321
PREEMPTIVE_OS_GETPROCADDRESS.....	321
What Is the PREEMPTIVE_OS_GETPROCADDRESS Wait Type?	322
PREEMPTIVE_OS_GETPROCADDRESS Example.....	324
Lowering PREEMPTIVE_OS_GETPROCADDRESS Waits.....	325
PREEMPTIVE_OS_GETPROCADDRESS Summary.....	325

Chapter 12: Background and Miscellaneous Wait Types.....	327
CHECKPOINT_QUEUE.....	328
What Is the CHECKPOINT_QUEUE Wait Type?	328
CHECKPOINT_QUEUE Summary.....	331
DIRTY_PAGE_POLL.....	332
What Is the DIRTY_PAGE_POLL Wait Type?.....	332
DIRTY_PAGE_POLL Summary	335
LAZYWRITER_SLEEP	335
What Is the LAZYWRITER_SLEEP Wait Type?	335
LAZYWRITER_SLEEP Summary	337
MSQL_XP	337
What Is the MSQL_XP Wait Type?	337
MSQL_XP Example	338
Lowering MSQL_XP Waits	339
MSQL_XP Summary	340
OLEDB	340
What Is the OLEDB Wait Type?.....	340
OLEDB Example	340
Lowering OLEDB Waits	341
OLEDB Summary	342
TRACEWRITE	342
What Is the TRACEWRITE Wait Type?	343
TRACEWRITE Example	343
Lowering TRACEWRITE Waits	348
TRACEWRITE Summary	351
WAITFOR	351
What Is the WAITFOR Wait Type?	352
WAITFOR Example	352
WAITFOR Summary.....	353

TABLE OF CONTENTS

Chapter 13: In-Memory OLTP–Related Wait Types.....	355
Introduction to In-Memory OLTP	355
CFPs	356
Isolation	360
Transaction Log Changes	360
WAIT_XTP_HOST_WAIT.....	361
What Is the WAIT_XTP_HOST_WAIT Wait Type?.....	361
WAIT_XTP_HOST_WAIT Summary	366
WAIT_XTP_CKPT_CLOSE.....	367
What Is the WAIT_XTP_CKPT_CLOSE Wait Type?	367
WAIT_XTP_CKPT_CLOSE Summary	370
WAIT_XTP_OFFLINE_CKPT_NEW_LOG.....	370
What Is the WAIT_XTP_OFFLINE_CKPT_NEW_LOG Wait Type?.....	370
WAIT_XTP_OFFLINE_CKPT_NEW_LOG Summary	372
Appendix I: Example SQL Server Machine Configurations.....	373
Default Test Machine.....	373
HA/DR Test Machines	374
Appendix II: Spinlocks.....	377
Appendix III: Latch Classes	381
Index.....	391

About the Author



Enrico van de Laar has been working with data in all kinds of formats and sizes for over 15 years. He is a Data & Advanced Analytics Consultant for Dataheroes where he helps organizations optimize their data platform environment and helps them with their first steps in the world of Advanced Analytics.

Enrico is a Data Platform MVP since 2014 and a frequent speaker on various data-related events all over the world.

He frequently writes about technologies, like Microsoft SQL Server and Azure Machine Learning, on his blog at www.enricovandelaar.com. You can contact Enrico through his Twitter handle @evdlaar or by sending him an e-mail at enrico@dataheroes.nl.

About the Technical Reviewers



Eelco Drost is a Data Platform Architect at Data Masterminds, a company that he co-formed in 2017.

He has over two decades of experience with SQL Server as a consultant DBA, programmer, and architect.

His areas of expertise are architecture, disaster recovery, database administration, database programming, performance tuning, and SQL Server Internals.

He speaks at user groups and other industry international events. You can follow him on Twitter at @eelcodrost, find him on LinkedIn at www.linkedin.com/in/eelcodrost, and drop him an e-mail at eelcodrost@datamasterminds.io.

Borbala Toth-Apathy is a database developer and architect with over 15 years of experience in the IT field. She got interested in performance tuning when starting to work with SQL Server back in the day, and it's still one of her favorite topics. Recently she is most fascinated by the power of data analysis—how raw data can transform into complete decision support systems and how this process can change the way people think about data.

Acknowledgments

First and foremost, I want to thank my family for supporting me while writing this book. They had some experience in that area from when I worked on the first edition of this book, but writing the second edition of this book still proved to be a big undertaking.

I want to thank Apress, especially Jonathan and Jill, for helping me write this book. When Jonathan asked me whether I was interested in updating the first edition, I could only answer with a “yes!”. So many things in SQL Server changed in the 4 years between the first edition and this book, and being able to update the book to reflect those changes means a lot to me.

A big thanks goes out to the technical editors, Eelco Drost and Borbala Toth-Apathy. The comments they provided helped clarify a lot of areas inside this book and resulted in a far higher quality.

Finally, a massive shout-out to the SQL Server community. You are a technical community that has no equal, and I am honored to be a part of it.

Introduction

Performance is a hot issue on a lot of database implementations. Many businesses run into performance-related issues when their databases experience more load or grow larger in size. There are many methods available for increasing the performance of your SQL Server(s) on all types of levels. Many of these performance-optimization methods we consider best practice, like running index maintenance to make sure fragmented indexes don't slow down your queries, or updating statistics so the SQL Server Database Engine has the correct information to generate a good execution plan. Besides these database maintenance methods, you can also choose to dive a little bit deeper into specific query performance troubleshooting, optimizing queries by making sure expensive operators are replaced by less expensive ones, for instance. And of course there is always the "sledgehammer" approach, replacing your current hardware for newer, better performing hardware, hoping that will solve the performance issues you are experiencing.

No matter what approach you choose to optimize or troubleshoot SQL Server performance, there are always two common resources involved: time and money. Ideally we want to spend as little time and money as possible while we are working on increasing performance. Knowing where to focus your time and money is very important. If you can find the source of the performance problem and resolve it at that level, you can save a lot of time and money that you would have spent on analyzing symptoms.

In a way, we can compare our search for the heart of our performance issues with a medical examination. Instead of giving out different types of medication until something actually works, a physician is always trying to find the source of the problem so he or she can prescribe the right medication that works best for that specific condition without causing side effects. The same approach works for SQL Server. Implementing all types of possible solutions without looking at the real source of the problem will probably not solve the real underlying issue (unless you're really lucky) and can possibly make matters worse.

INTRODUCTION

This is where wait statistics can help. Wait statistics are generated and maintained at the heart of the SQL Server Database Engine where queries are being executed, giving valuable insight into what is slowing down your queries. There are 921 different types of wait statistics in the latest edition of SQL Server (SQL Server 2017), and with every edition that number grows as new features are introduced or existing features are modified or expanded. That is a lot of information that is freely available to help you troubleshoot!

This book is my attempt to help you understand SQL Server wait statistics. It will go into detail how wait statistics are being generated and how you can use that information to optimize, or troubleshoot, the performance of your SQL Server installation. I will also describe specific wait statistics and give you pointers on how you can resolve problems yourself. In the case of this book, I personally believe the journey is more important than reaching the destination. For that reason, I spend more time describing and explaining what is causing the specific wait types to occur than I do writing down every possible way you can lower their wait times. If you understand why a wait type is generated, and to what part of SQL Server it is related, you will quickly discover methods of your own to lower their wait times.

Because of the sheer number of different wait statistics, it is sadly impossible to describe and discuss all of them. For this reason, I had to make a selection of wait statistics to include in this book. The way I did this was by gathering wait statistics information from many different SQL Server installations and selecting the most common or most performance-degrading ones, resulting in a selection of 45 different (or grouped) wait types.

Book Layout

As I wrote in the introduction, the goal of this book is to give you a deeper understanding of SQL Server wait statistics and also to describe various wait statistics in detail. For this reason this book has been split up into two parts, Part I describing the foundation of wait statistics analysis and Part II describing various specific wait statistics in detail. I tried to categorize the wait statistics in Part II by the part of the system they affect (i.e., CPU, Memory, etc.). Some wait statistics aren't that easily categorized, since they can affect multiple system parts. In those cases, I tried to categorize them with the part they have the most effect on.

Part I: Foundations of Wait Statistics Analysis

- Chapter 1: “Wait Statistics Internals” starts off with a brief history of SQL Server wait statistics and a look at the SQLOS architecture. Because wait statistics have a close relationship with the processor(s) of your system, we will discuss schedulers, tasks, and worker threads in detail.
- Chapter 2: “Querying SQL Server Wait Statistics” introduces the various ways you can access the wait statistics information inside the SQL Server Database Engine by using DMVs, Extended Events, and Perfmon.
- Chapter 3: “The Query Store” explores a new SQL Server feature that became available in SQL Server 2016, the Query Store. Since the Query Store can have a massive impact on how you perform performance analysis, I dedicated a chapter on how you can use this amazing new feature.
- Chapter 4: “Building a Solid Baseline” covers the importance of building and using a baseline for performance troubleshooting. Baselines are especially important when analyzing wait statistics, and I will show you examples of how you can create one for wait statistics analysis.

Part II: Wait Types

- Chapter 5: “CPU-Related Wait Types” introduces wait types that are CPU related.
- Chapter 6: “IO-Related Wait Types” describes wait types that are IO related.
- Chapter 7: “Backup-Related Wait Types” presents the wait types that are related to backup events.
- Chapter 8: “Lock-Related Wait Types” starts off with a short introduction to locking and blocking in SQL Server before diving into the lock-related wait types.

INTRODUCTION

- Chapter 9: “Latch-Related Wait Types” starts off with a close look at latches, describing what they are and how they work inside SQL Server. After this introduction we are ready to take a look at different latch-related wait types.
- Chapter 10: “High-Availability and Disaster-Recovery Wait Types” describes wait types that are related to the various HA and DR configurations available in SQL Server.
- Chapter 11: “Preemptive Wait Types” presents different wait types that have a direct relationship with the operating system of your SQL Server.
- Chapter 12: “Background and Miscellaneous Wait Types” includes various wait types that are being generated by the SQLOS as a background process. This is also the chapter where we will describe wait types that couldn’t be categorized in one of the preceding categories.
- Chapter 13: “In-Memory OLTP-Related Wait Types” describes some of the wait types that are related to the In-Memory OLTP feature which was released in SQL Server 2014.
- Appendix I: “Example SQL Server Machine Configurations” describes the configuration of the virtual machines I used in the book.
- Appendix II: “Spinlocks” explains the working of so-called lightweight synchronization primitives called spinlocks and the impact they can have on your SQL Server configuration.
- Appendix III: “Latch Classes” contains a list of a large portion of different latch classes inside SQL Server. The list is a combination of information from Books Online and additional information about the specific latch class.

Word of Warning

The way we access wait statistics information in this book is mostly by SQL queries. For this reason this book has quite a lot of lines of SQL code. Most of the queries in Part I of this book deal with gathering or capturing wait statistics using Dynamic Management Views (or DMVs) and as such are, unless stated otherwise, harmless. Some of the queries in Part II, however, are written to actually harm performance so specific wait types can be demonstrated. Please keep this in mind when you plan to use some of the scripts and test them thoroughly on a test system.

PART I

Foundations of Wait Statistics Analysis

CHAPTER 1

Wait Statistics Internals

SQL Server wait statistics are an important tool you can use to analyze performance-related problems or to optimize your SQL Server's performance. They are, however, not that well known to many database administrators or developers. I believe this has to do with their relatively complex nature, the sheer volume of the different types of wait statistics, and the lack of documentation for many types of wait statistics. Wait statistics are also directly related to the SQL Server you are analyzing them on, which means that it is impossible to compare the wait statistics of Server A to the wait statistics of Server B, even if they had an identical hardware and database configuration. Every configuration option, from the hardware firmware level to the configuration of the SQL Server Native Client on the client computers, will have an impact on the wait statistics!

For the reasons just mentioned, I firmly believe we should start with the foundation and internals of SQL Server wait statistics so you can get familiar with how they are generated, how you can access them, and how you can use them for performance troubleshooting. This approach will get you ready for Part II of this book, where we will examine specific wait statistics.

In this chapter we will take a brief look at the history of wait statistics through the various versions of SQL Server. Following that, we will take a close look at the SQL Operating System, or SQLOS. The architecture of the SQLOS is closely tied to wait statistics and to performance troubleshooting in general. The rest of the chapter is dedicated to one of the most important aspects of wait statistics: thread scheduling.

Before we begin with the foundation and internals of SQL Server wait statistics, I would like to mention a few things related to the terminology used when discussing wait statistics. In the introduction of this book and the preceding paragraphs, I only mentioned the term wait statistics. The sentence “compare the wait statistics of Server A to the wait statistics of Server B” is actually wrong, since we can only compare the *wait time* (the total time we have been waiting on a resource) of a specific *wait type*

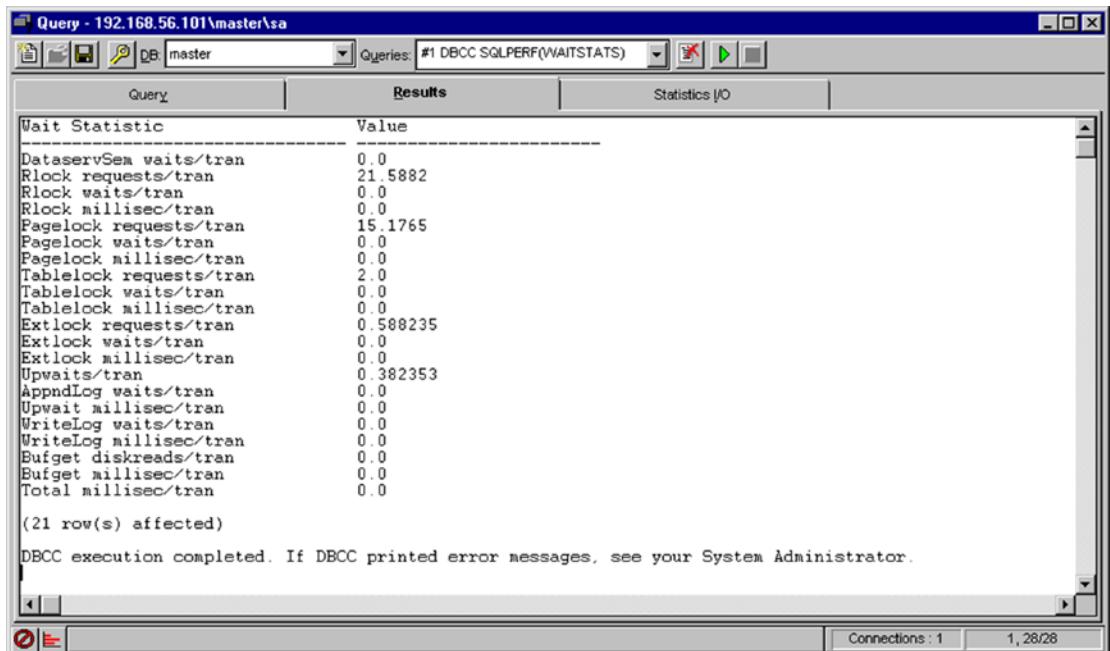
(the specific wait type related to the resource we are waiting on). From this point on, when I use the term *wait statistics* I mean the concept of wait statistics, and I will use the correct terms wait time and wait type where appropriate.

A Brief History of Wait Statistics

SQL Server has been around for quite some time now; the first release of SQL Server dates back to 1989 and was released for the OS/2 platform. Until SQL Server 6.0, released in 1995, Microsoft worked together with Sybase to develop SQL Server. In 1995, however, Microsoft and Sybase went their own ways. Microsoft and Sybase stayed active in the database world (SAP actually acquired Sybase in 2010), and in 2019 Microsoft will release SQL Server 2019 while SAP released SAP Sybase ASE 16 in 2014 (but is still maintained today), both relational enterprise-level database systems.

Between SQL Server 6.0 and SQL Server 2019, so many things have changed that you simply cannot compare the two versions. One thing that hasn't changed in all these years is wait statistics. In one way or another, SQL Server stores information about its internal processes, and even though the way we access that information has changed over the years, wait statistics remain an important part of the internal logging process.

In early versions of SQL Server we needed to access the wait statistics using undocumented commands. Figure 1-1 shows how you would query wait statistics information in SQL Server 6.5 using the DBCC command.



The screenshot shows a SQL Server Management Studio window titled "Query - 192.168.56.101\master\sa". The query results pane displays the output of the DBCC SQLPERF(WAITSTATS) command. The results are presented in a table with two columns: "Wait Statistic" and "Value". The "Value" column contains mostly 0.0 entries, except for "Rlock requests/tran" which is 21.5882, "Page lock requests/tran" which is 15.1765, and "Extlock requests/tran" which is 0.588235. A message at the bottom states "(21 row(s) affected)" and "DBCC execution completed. If DBCC printed error messages, see your System Administrator.".

Wait Statistic	Value
DataservSem waits/tran	0.0
Rlock requests/tran	21.5882
Rlock waits/tran	0.0
Rlock millisec/tran	0.0
Page lock requests/tran	15.1765
Page lock waits/tran	0.0
Page lock millisec/tran	0.0
Table lock requests/tran	2.0
Table lock waits/tran	0.0
Table lock millisec/tran	0.0
Ext lock requests/tran	0.588235
Ext lock waits/tran	0.0
Ext lock millisec/tran	0.0
Up wait/tran	0.382353
AppndLog waits/tran	0.0
Up wait millisec/tran	0.0
WriteLog waits/tran	0.0
WriteLog millisec/tran	0.0
Bufilet disk reads/tran	0.0
Bufilet millisec/tran	0.0
Total millisec/tran	0.0

(21 row(s) affected)
DBCC execution completed. If DBCC printed error messages, see your System Administrator.

Figure 1-1. SQL Server wait statistics in SQL Server 6.5

One of the big changes that were introduced in SQL Server 2005 was the conversion of many internal functions and commands into Dynamic Management Views (DMVs), including wait statistics information. This made it far easier to query and analyze the information returned by functions or commands. A new way of performance analysis was born with the release of the SQL Server 2005 Microsoft whitepaper “SQL Server 2005 Waits and Queues” by Tom Davidson.

In the various releases of SQL Server the amount of different wait types grew exponentially whenever new features or configuration options were introduced. If you take a good look at Figure 1-1 you will notice that 21 different wait types were returned. Figure 1-2 shows the amount of wait types, as the number of rows returned, available in SQL Server 2017.

The screenshot shows a SQL Server Management Studio window with a query results grid. The query is:

```
SELECT * FROM sys.dm_os_wait_stats
```

The results grid has the following columns:

wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
MISCELLANEOUS	0	0	0	0
LCK_M_SCH_S	0	0	0	0
LCK_M_SCH_M	0	0	0	0
LCK_M_S	8	1493	260	2
LCK_M_U	0	0	0	0
LCK_M_X	0	0	0	0
LCK_M_IS	0	0	0	0
LCK_M_IU	0	0	0	0
LCK_M_IX	0	0	0	0
LCK_M_SIU	0	0	0	0
LCK_M SIX	0	0	0	0

Below the grid, a message bar indicates "Query executed successfully." and shows connection details: EVDL-SQL2017-01 (14.0 RTM) | EVDL-SQL2017-01\Administr... | master | 00:00:00 | 921 rows.

Figure 1-2. SQL Server wait statistics in SQL Server 2017

Those 921 rows are all different wait types and hold wait information for different parts of the SQL Server engine. With the release of SQL Server 2019 Community Technology Preview (CTP) 2.4, the number of wait types increased even further and cross the line of more than 1.000 different wait types. The number of wait types will continue to grow in future SQL Server releases, as new features are introduced or existing features are changed. Thankfully there is a lot more information available about wait statistics now than there was in SQL Server 6.5!

The SQLOS

The world of computer hardware changes constantly. Every year, or in some cases every month, we manage to put more cores inside a processor, increase the memory capacity of mainboards, or introduce entirely new hardware concepts like PCI-based persistent flash storage. Database Management Systems (or DBMSs) are always one of the first types of applications that want to take advantage of new hardware trends. Because of the fast-changing nature of hardware and the need to utilize new hardware options as soon as they become available, the SQL Server team decided to change the SQL Server platform layer in SQL Server 2005.

Before SQL Server 2005, the platform layer of SQL Server was pretty restricted, and many operations were performed by the operating system. This meant that it was difficult for SQL Server to keep up with the fast-changing world of server hardware, as changing a complete operating system in order to utilize faster hardware or new hardware features is a time-consuming and complex operation.

Figure 1-3 shows the (simplified) architecture of SQL Server before the introduction of the SQLOS in SQL Server 2005.

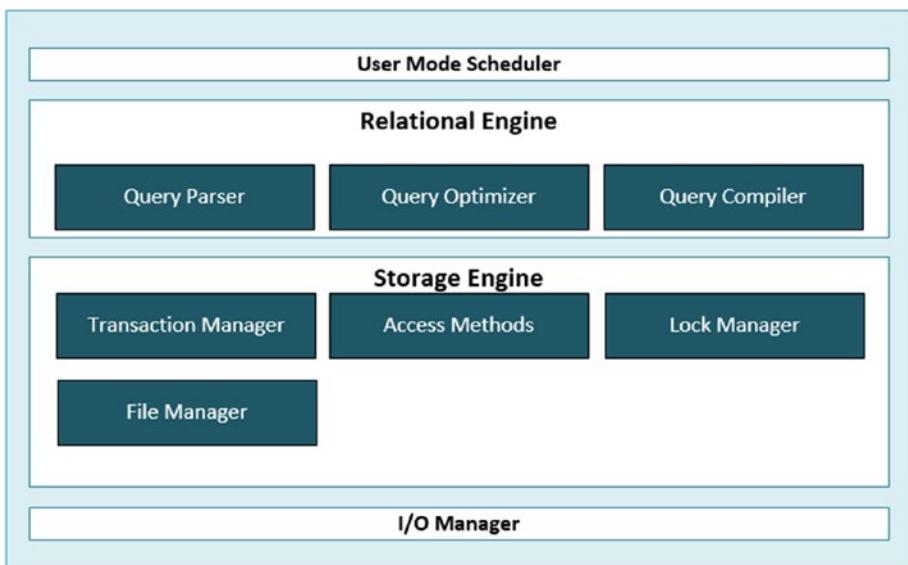


Figure 1-3. SQL Server architecture before the introduction of the SQLOS

SQL Server 2005 introduced one of the biggest changes to the SQL Server engine seen to this day, the SQLOS. This is a completely new platform layer that functions as a user-level operating system. This new operating system has made it possible to fully utilize current and future hardware and has enabled features like advanced parallelism. The SQLOS is highly configurable and adjusts itself to the hardware it is running on, thus making it perfectly scalable for high-end or low-end systems alike.

Figure 1-4 shows the (simplified) architecture of SQL Server 2005, including the SQLOS layer.



Figure 1-4. SQL Server 2005 architecture

The SQLOS changed the way SQL Server accesses processor resources by introducing schedulers, tasks, and worker threads. This gives the SQLOS greater control of how work should be completed by the processors. The Windows operating system uses a *preemptive scheduling* approach. This means that Windows will give every process that needs processor time a priority and fixed slice of time, or a *quantum*. This process priority is calculated from a number of variables like resource usage, expected runtime, current activity, and so forth. By using preemptive scheduling, the Windows operating system can choose to interrupt a process when a process with a higher priority needs processor time. This way of scheduling can have a negative impact on processes generated by SQL Server, since those processes could easily be interrupted by higher priority ones, including those of other applications. For this reason, the SQLOS uses its own (cooperative) non-preemptive scheduling mechanism, making sure that Windows processes cannot interrupt SQLOS processes.

SQL Server 7 and SQL Server 2000 also used non-preemptive scheduling using User Mode Scheduling (UMS). SQLOS brought many more system components closer together, thus enabling better performance and scalability.

There are some exceptions when the SQLOS cannot use non-preemptive scheduling, for instance, when the SQLOS needs to access a resource through the Windows operating system. We will discuss these exceptions later in this book in Chapter 11, “Preemptive Wait Types.”

Schedulers, Tasks, and Worker Threads

Because the SQLOS uses a different method to execute requests than the Windows operating system uses, SQL Server introduced a different way to schedule processor time using schedulers, tasks, and worker threads. Figure 1-5 shows the different parts of SQL Server scheduling and how they relate to each other.

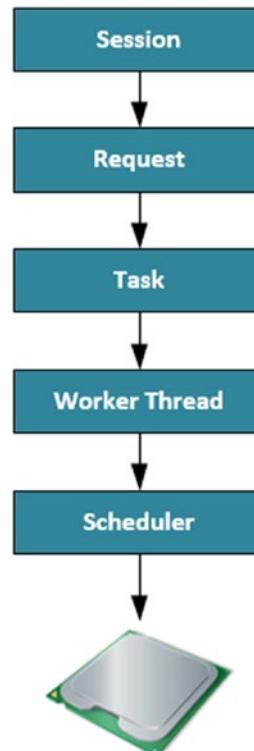


Figure 1-5. SQL Server scheduling

Sessions

A *session* is the connection a client has to the SQL Server it is connected to (after it has been successfully authenticated). We can easily access session information by querying the **sys.dm_exec_sessions** DMV using the following query:

```
SELECT * FROM sys.dm_exec_sessions;
```

Generally speaking, user sessions will have a `session_id` greater than 50; everything lower is reserved for internal SQL Server processes. However, on very busy servers there is a possibility that SQL Server needs to use a `session_id` greater than 50. If you are only interested in information about user-initiated sessions, it is better to filter the results of the `sys.dm_exec_sessions` DMV using the `is_user_process` column instead of filtering on a `session_id` greater than 50. The following query will only return user sessions and will filter out the internal system sessions:

```
SELECT * FROM sys.dm_exec_sessions
WHERE is_user_process = 1;
```

Figure 1-6 shows a small part of the results of this query.

	session_id	login_time	host_name	program_name	host_process_id	client_version
1	51	2019-01-24 19:21:23.567	EVDL-SQL2017-01	SQLServerCEIP	4784	7
2	53	2019-01-24 19:22:05.473	EVDL-SQL2017-01	Microsoft SQL Server Management Studio	4652	7
3	54	2019-01-24 19:22:05.440	EVDL-SQL2017-01	Microsoft SQL Server Management Studio	4652	7
4	55	2019-01-24 19:23:11.003	EVDL-SQL2017-01	Microsoft SQL Server Management Studio - Query	4652	7

Figure 1-6. *sys.dm_exec_sessions* results

There are many more columns returned by the `sys.dm_exec_sessions` DMV that will give us information about the specific session. Some of the more interesting columns that deserve some extra explanation are the `host_process_id`, which is the Process ID (or PID) of the client program connected to the SQL Server. The `cpu_time` column will give you information about the processor time (in milliseconds) the session has used since it was first established. The `memory_usage` column displays the amount of memory used by the session. This is not the amount in MB or KB, but the number of 8 KB pages used. Another column I would like to highlight is the `status` column. This will show you if the session has any active requests. The most common values of the `status` column

are “Running,” which indicates that one or more requests are currently being processed from this session, and “Sleeping,” which means no requests are currently being processed from this session.

Requests

A *request* is the SQL Server execution engine’s representation of a query submitted by a session. Again, we can use a DMV to query information about a request; in this case, we can run a query against the **sys.dm_exec_requests** DMV like the following query:

```
SELECT * FROM sys.dm_exec_requests;
```

Figure 1-7 shows a portion of the results of this query.

	session_id	request_id	start_time	status	command	sql_handle
25	25	0	2019-01-24 19:24:20.787	sleeping	TASK MANAGER	NULL
26	26	0	2019-01-24 19:24:20.787	sleeping	TASK MANAGER	NULL
27	27	0	2019-01-24 19:24:20.787	sleeping	TASK MANAGER	NULL
28	28	0	2019-01-24 19:24:20.787	sleeping	TASK MANAGER	NULL
29	29	0	2019-01-22 10:15:38.823	background	HADR_AR_MGR_NOTIFICATION...	NULL
30	30	0	2019-01-24 19:20:05.200	sleeping	TASK MANAGER	NULL
31	31	0	2019-01-24 19:24:50.863	sleeping	TASK MANAGER	NULL
32	32	0	2019-01-22 10:15:39.167	background	BRKR EVENT HNDLR	NULL
33	33	0	2019-01-22 10:15:39.167	background	BRKR TASK	NULL
34	42	0	2019-01-22 10:15:39.167	background	BRKR TASK	NULL
35	43	0	2019-01-22 10:15:39.167	background	BRKR TASK	NULL
36	50	0	2019-01-24 19:21:05.257	sleeping	TASK MANAGER	NULL
37	55	0	2019-01-24 19:25:03.973	running	SELECT	0x02000...

Figure 1-7. *sys.dm_exec_requests* results

The **sys.dm_exec_requests** DMV is an incredibly powerful tool to use when you are troubleshooting any performance-related issues. The reason for this is that it has a lot of information about the actual queries being executed and can help you detect performance bottlenecks relatively quickly. Because the **sys.dm_exec_requests** DMV also displays wait statistics-related information, we will take a thorough look at it in Chapter 2, “Querying SQL Server Wait Statistics.”

Tasks

Tasks represent the actual work that needs to be performed by the SQLOS, but they do not perform any work themselves. When a request is received by SQL Server, one or more tasks will be created to fulfill the request. The number of tasks that get generated for a request depends on if the query request is being performed using parallelism or if it's being run serially.

We can use the **sys.dm_os_tasks** DMV to query the task information, like I did in the following example query:

```
SELECT * sys.dm_os_tasks;
```

Figure 1-8 shows a part of the results of the query.

	task_address	task_state	context_switches_count	pending_io_count	pending_io_byte_count	pending_io_byte_average	scheduler_id
1	0x0000007F7D01C4E8	SUSPENDED	42911	0	0	0	0
2	0x0000007F7D01C8C8	RUNNING	4	0	0	0	0
3	0x0000007F7D01CCA8	RUNNING	32	0	0	0	0
4	0x0000007F7D01D468	RUNNING	3	0	0	0	0
5	0x0000007F7D01D848	DONE	NULL	NULL	NULL	NULL	0
6	0x0000007F7D01DC28	SUSPENDED	615	515	0	0	0
7	0x0000007F78B5C108	SUSPENDED	140339	0	0	0	0
8	0x0000007F78B5C4E8	SUSPENDED	413	51	0	0	0
9	0x0000007F78B5C8C8	SUSPENDED	132339	0	0	0	0
10	0x0000007F78B5D088	DONE	NULL	NULL	NULL	NULL	0
11	0x0000007F78B5CCA8	SUSPENDED	1	0	0	0	0
12	0x0000007F7453E8C8	SUSPENDED	1	0	0	0	0

Figure 1-8. *sys.dm_os_tasks* results

When you query the **sys.dm_os_tasks** DMV you will discover it will return many results, even on servers that have no user activity. This is because SQL Server uses tasks for its own processes as well; you can identify those by looking at the **session_id** column.

There are some interesting columns in this DMV that are worth exploring to see the relations between the different DMVs. The **task_address** column will show you the memory address of the task. The **session_id** will return the ID of the session that has requested the task, and the **worker_address** will hold the memory address of the worker thread associated with the task.

Worker Threads

Worker threads are where the actual work for the request is being performed. Every task that gets created will get a worker thread assigned to it, and the worker thread will then perform the actions requested by the task.

A worker thread will actually not perform the work itself; it will request a thread from the Windows operating system to perform the work for it. For the sake of simplicity, and the fact the actual Windows thread runs outside the SQLOS, I have left this step out of Figure 1-5. You can access information about the Windows operating system threads by querying **sys.dm_os_threads** if you are interested.

When a task requests a worker thread SQL Server will look for an idle worker thread and assign it to the task. In the case when no idle worker thread can be located and the maximum number of worker threads has been reached, the request will be queued until a worker thread finishes its current work and becomes available.

There is a limit to the number of worker threads SQL Server has available for processing requests. This number will be automatically calculated and configured by SQL Server during startup. We can also calculate the maximum number of worker threads ourselves using these formulas:

- 32-bit system with less than, or equal to, 4 logical processors:
 - 256 worker threads
- 32-bit system with more than 4 logical processors:
 - $256 + ((\text{number of logical processors} - 4) * 8)$
- 64-bit system with less than, or equal to, 4 logical processors:
 - 512 worker threads
- 64-bit system with more than 4 logical processors:
 - $512 + ((\text{number of logical processors} - 4) * 16)$

Example: If we have a 64-bit system with 16 processors (or cores) we can calculate the maximum number of worker threads using the formula, $512 + ((16 - 4) * 16)$, which would give us a maximum of 704 worker threads.

The number of worker threads can be changed from the default of **0** (which means SQL Server sets the number of max worker threads using the preceding formulas when it starts) by changing the **max worker threads** options in your SQL Server's properties, as illustrated by Figure 1-9.

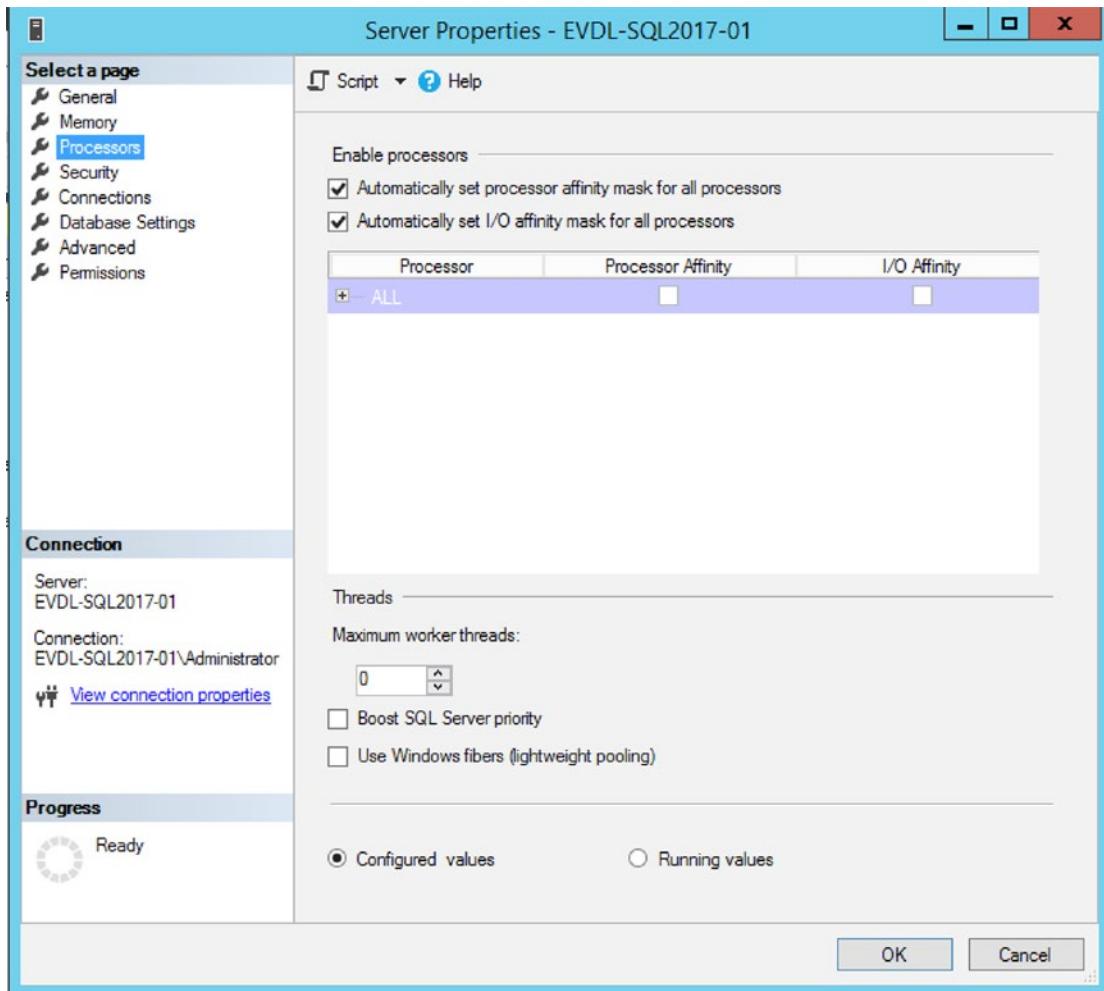


Figure 1-9. Processors page in the Server Properties

Generally speaking, there should be no need to change the **max worker threads** option, and my advice is to leave the setting alone, as it should only be changed in very specific cases (I will discuss one of those potential cases in Chapter 5, “CPU-Related Wait Types,” when we talk about THREADPOOL waits).

One thing to keep in mind is that worker threads require memory to work. For 32-bit systems this is 512 KB for every worker thread; 64-bit systems will need 2048 KB for every worker thread. Thus, changing the number of worker threads can potentially impact the memory requirements of SQL Server. This does not mean you need a massive amount of memory just for your worker threads—SQL Server will automatically destroy worker threads if they have been idle for 15 minutes or if your SQL Server is under heavy memory pressure.

SQL Server supplies us with a DMV to query information about the worker threads: **sys.dm_os_workers**. Figure 1-10 shows some of the results of this query:

```
SELECT * FROM sys.dm_os_workers;
```

	worker_address	status	is_preemptive	is_fiber	is_sick	is_in_cc_exception	is_fatal_exception	is_inside_catch
1	0x00000007F7D020160	2	0	0	0	0	0	0
2	0x00000007F7D022160	4	1	0	0	0	0	0
3	0x00000007F7D02A160	2	0	0	0	0	0	0
4	0x00000007F7D200160	2097156	1	0	0	0	0	0
5	0x00000007F7D21C160	4	1	0	0	0	0	0
6	0x00000007F7D21E160	4	1	0	0	0	0	0
7	0x00000007F7D258160	0	0	0	0	0	0	0
8	0x00000007F7D064160	4	1	0	0	0	0	0
9	0x00000007F7A62C160	0	0	0	0	0	0	0
10	0x00000007F78A8E160	0	0	0	0	0	0	0
11	0x00000007F78A90160	4	1	0	0	0	0	0
12	0x00000007F78A92160	0	0	0	0	0	0	0

Figure 1-10. Results of querying sys.dm_os_workers

The **sys.dm_os_workers** DMV is a very large and complex DMV where many columns are marked as “Internal use only” by Microsoft. In this DMV the columns **task_address** and **scheduler_address** are available to link together the different DMVs we have discussed.

Worker threads go through different phases while they are being exposed to the processor, which we can view when we look at the **state** column in the **sys.dm_os_workers** DMV:

- INIT: The worker thread is being initialized by the SQLOS.
- RUNNING: The worker thread is currently performing work on a processor.

- **RUNNABLE:** The worker thread is ready to run on a processor.
- **SUSPENDED:** The worker thread is waiting for a resource.

The states the worker threads go through while performing their work are one of the main topics of this book. Every time a worker thread is not in the “RUNNING” state, it has to wait, and the SQLOS records this information into wait statistics, giving us valuable insight into what the worker thread has been waiting on and how long it has been waiting.

Schedulers

The *scheduler* component’s main task is to—surprise—schedule work, in the form of tasks, on the physical processor(s). When a task requests processor time it is the scheduler that assigns worker threads to that task so the request can get processed. It is also responsible for making sure worker threads cooperate with each other and yield the processor when their slice of time, or quantum, has expired. We call this *cooperative scheduling*. The need for worker threads to yield when their processor time has expired comes from the fact that a scheduler will only let one worker thread run on a processor at a time. If the worker threads didn’t need to yield, a worker thread could stay on the processor for an infinite amount of time, blocking all usage of that processor.

There is a one-on-one relation between processors and schedulers. If your system has two processors, each with four cores, there will be eight schedulers that the SQLOS can use to process user requests, each of them mapped to one of the logical processors.

We can access information about the schedulers by running a query against the **sys.dm_osSchedulers** DMV:

```
SELECT * FROM sys.dm_osSchedulers;
```

The results of the query are shown in Figure 1-11.

	scheduler_address	parent_node_id	scheduler_id	cpu_id	status	is_online	is_idle	preemptive_switches_count
1	0x00000007F7D180040	0	0	0	VISIBLE ONLINE	1	0	37589
2	0x00000007F7D1A0040	0	1	1	VISIBLE ONLINE	1	1	59238
3	0x00000007F7D1C0040	0	1048578	0	HIDDEN ONLINE	1	0	0
4	0x00000007F7D800040	64	1048576	0	VISIBLE ONLINE (DAC)	1	1	4
5	0x00000007F78960040	0	1048579	1	HIDDEN ONLINE	1	1	0
6	0x00000007F7D1E0040	0	1048580	0	HIDDEN ONLINE	1	1	5
7	0x00000007F78940040	0	1048581	1	HIDDEN ONLINE	1	1	0
8	0x00000007F76000040	0	1048582	0	HIDDEN ONLINE	1	1	2

Figure 1-11. *sys.dm_osSchedulers* query results

The SQL Server on which I ran this query has one processor with two cores, which means there should be two schedulers that can process my user requests. If we look at Figure 1-11, however, we notice there are more than two schedulers returned by the query. SQL Server uses its own schedulers to perform internal tasks, and those schedulers are also returned by the DMV and are marked “HIDDEN ONLINE” in the status column of the DMV. The schedulers that are available for user requests are marked as “VISIBLE ONLINE” in the DMV. There is also a special type of scheduler with the status “VISIBLE ONLINE (DAC).” This is a scheduler dedicated for use with the Dedicated Administrator Connection (DAC). This scheduler makes it possible to connect to SQL Server in situations where it is unresponsive; for instance, when there are no free worker threads available on the schedulers that process user requests.

We can view the number of worker threads a scheduler has associated with it by looking at the `current_workers_count` column. This number also includes worker threads that aren’t performing any work. The `active_workers_count` shows us the worker threads that are active on the specific scheduler. This doesn’t mean they are actually running on the processor, as worker threads with states of “RUNNING,” “RUNNABLE,” and “SUSPENDED” also count toward this number. The `work_queue_count` is also an interesting column since it will give you insight into how many tasks are waiting for a free worker thread. If you see high numbers in this column, it might mean that you are experiencing CPU pressure.

Putting It All Together

All the parts of the SQL Server scheduling we have discussed so far are connected to each other, and every request passes through these same components. The following text is an example of how a query request would get processed.

A user connects to the SQL Server through an application. The SQL Server will create a session for that user after the login process is completed successfully. When the user sends a query to the SQL Server, a task and a request will be created to represent the unit of work that needs to be done. The scheduler will assign worker threads to the task so it can be completed.

To see all this information in SQL Server, we can join some of the DMVs we used. The query in Listing 1-1 will show you an example of how we can combine the different DMVs to get scheduling information about a specific session (in this case a session with an ID of 55).

Listing 1-1. Join the different DMVs together to query scheduling information

```

SELECT
    r.session_id AS 'Session ID',
    r.command AS 'Type of Request',
    qt.[text] AS 'Query Text',
    t.task_address AS 'Task Address',
    t.task_state AS 'Task State',
    w.worker_address AS 'Worker Address',
    w.[state] AS 'Worker State',
    s.scheduler_address AS 'Scheduler Address',
    s.[status] AS 'Scheduler State'

FROM sys.dm_exec_requests r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) qt
INNER JOIN sys.dm_os_tasks t
ON r.task_address = t.task_address
INNER JOIN sys.dm_os_workers w
ON t.worker_address = w.worker_address
INNER JOIN sys.dm_osSchedulers s
ON w.scheduler_address = s.scheduler_address
WHERE r.session_id = 55

```

Figure 1-12 shows the information that the query returned on my test SQL Server. To keep the results readable, I only selected columns from the DMVs to show the relation between them.

	Session ID	Type of Request	Query Text	Task Address	Task State	Worker Address	Worker State	Scheduler Address
1	55	SELECT	SELECT r.session_id AS 'Session ID', ...	0x0000007F7453F848	RUNNING	0x0000007F701CA160	RUNNING	0x0000007FD180040

Figure 1-12. Results of the query from Listing 1-1

In the results we can see that Session ID 53 made a SELECT query request. I did a cross apply with the **sys.dm_exec_sql_text** Dynamic Management Object to show the query text of the request. The request was mapped to a task, and the task began running. The task was then mapped to a worker thread that was then also in a running state. This meant that this query began being processed on a processor. The Scheduler Address column shows on which specific scheduler our worker thread was being run.

Wait Statistics

So far, we have gone pretty deep into the different components that perform scheduling for SQL Server and how they are interconnected, but we haven't given a lot of attention to the topic of this book: wait statistics.

In the section about worker threads earlier in this chapter, I described the states a worker thread can be in while it is performing work on a scheduler. When a worker thread is performing its work, it goes through three different phases (or queues) in the scheduler process. Depending on the phase (or queue) a worker thread is in, it will get either the "RUNNING," "RUNNABLE," or "SUSPENDED" state. Figure 1-13 shows an abstract view of a scheduler with the three different phases.

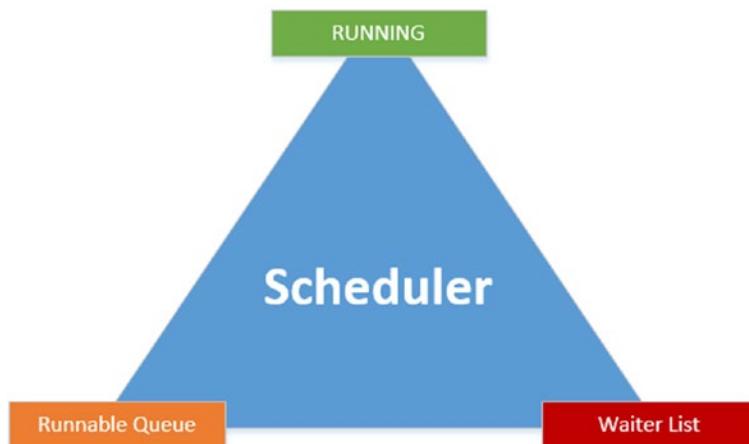


Figure 1-13. Scheduler and its phases and queues

When a worker thread gets access to a scheduler it will generally start in the Waiter List and get the "SUSPENDED" state. The *Waiter List* is an unordered list of worker threads that have the "SUSPENDED" state and are waiting for resources to become available. Those resources can be just about anything on the system, from data pages to a lock request. While a worker thread is in the Waiter List the SQLOS records the type of resource it needs to continue its work (the wait type) and the time it spends waiting before that specific resource becomes available, known as the *resource wait time*.

Whenever a worker thread receives access to the resources it needs, it will move to the *Runnable Queue*, a first-in-first-out list of all the worker threads that have access to their resources and are ready to be run on the processor. The time a worker thread spends in the Runnable Queue is recorded as the *signal wait time*.

The first worker thread in the Runnable Queue will move to the “RUNNING” phase, where it will receive processor time to perform its work. The time it spends on the processor is recorded as *CPU time*. In the meantime, the other worker threads in the Runnable Queue will move a spot higher in the list, and worker threads that have received their requested resources will move from the Waiter List into the Runnable Queue.

While a worker thread is in the “RUNNING” phase there are three scenarios that can happen:

- The worker thread needs additional resources; in this case it will move from the “RUNNING” phase to the Waiter List.
- The worker thread spends its quantum (fixed value of 4 milliseconds) and has to yield; the worker thread is moved to the bottom of the Runnable Queue.
- The worker thread is done with its work and will leave the scheduler.

Worker threads move through the three different phases all the time, and it is very common that one worker thread moves through them multiple times until its work is done.

Figure 1-14 will show you the scheduler view from Figure 1-13 combined with the different types of wait time and the flow of worker threads.

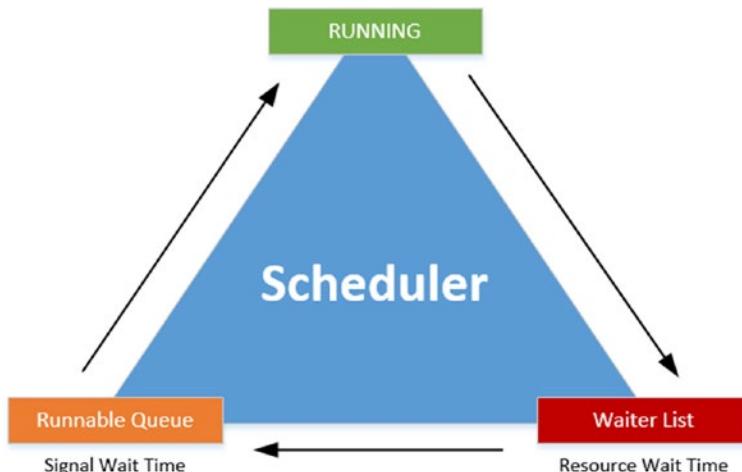


Figure 1-14. Scheduler view complete with wait times and worker thread flow

Knowing all the different lengths of time a request spends in one of the three different phases makes it possible to calculate the total request execution time, and also the total time a request had to wait for either processor time or resource time. Figure 1-15 shows the calculation of the total execution time and its different parts.

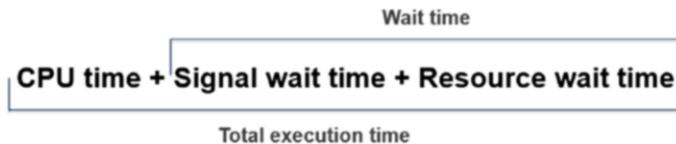


Figure 1-15. Request execution time calculation

Since there is a lot of terminology involved into the scheduling of worker threads in SQL Server, I would like to give you an example on how worker threads move through a scheduler.

Figure 1-16 will show you an abstract image of a scheduler like those we have already looked at, but this time I added requests that are being handled by that scheduler.

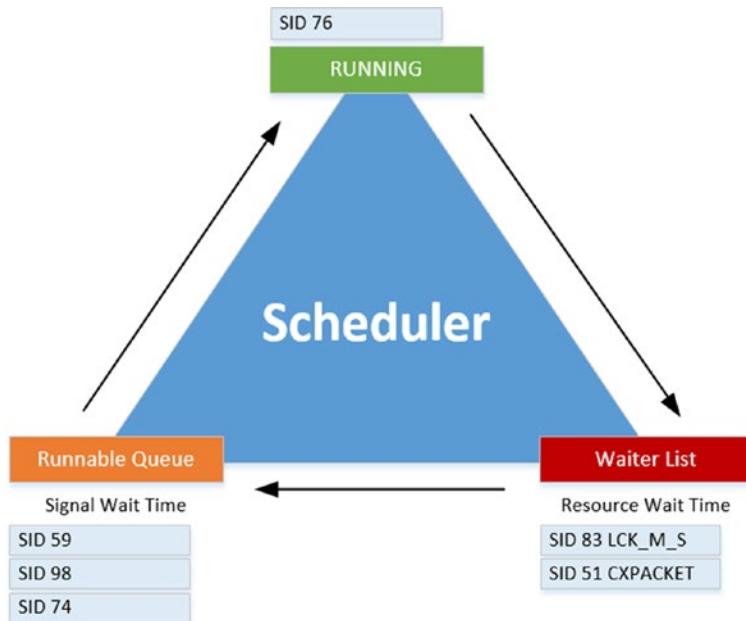


Figure 1-16. Scheduler with running requests

In this example we see that the request from **SID (Session ID) 76** is currently being executed on the processor; this request will have the state “RUNNING.” There are two other requests, **SID 83** and **SID 51**, in the Waiter List waiting for their requested resources. The wait types they are waiting for are LCK_M_S and CXPACKET. I won’t go into detail here about these wait types since we will be covering both of them in Part II of this book. While these two sessions are in the Waiter List, SQL Server will be recording the time they spend there as wait time, and the wait type will be noted as the representation of the resource they are waiting on. If we were to query information about these two threads, they would both have the “SUSPENDED” state. **SID 59**, **SID 98**, and **SID 74** have their resources ready and are waiting in the Runnable Queue for **SID 76** to complete its work on the processor. While they are waiting in the Runnable Queue, SQL Server records the time they spend there as the signal wait time and adds that time to the total wait time. These three worker threads will have the status of “RUNNABLE.”

In Figure 1-17 we have moved a few milliseconds forward in time; notice how the scheduler and worker threads have moved through the different phases and queues.

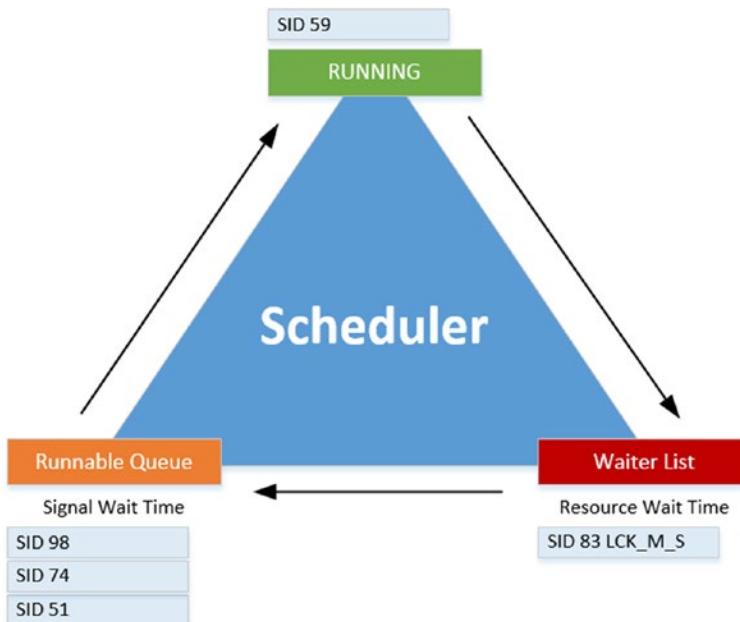


Figure 1-17. Scheduler a few milliseconds later

SID 76 completed its time on the processor; it didn't need any additional resources to complete its request and thus left the scheduler. **SID 59** was the first worker thread in the Runnable Queue, and now that the processor is free it will move from the Runnable Queue to the processor, and its state will change from "RUNNABLE" to "RUNNING." **SID 51** is done waiting on the CXPACKET wait type and moved from the Waiter List to the bottom of the Runnable Queue, changing its state from "SUSPENDED" to "RUNNABLE."

Summary

In this chapter we took a look at the history of wait statistics throughout various versions of SQL Server. Even though the method of analyzing SQL Server performance using wait statistics is relatively new, wait statistics have been a part of the SQL Server engine for a very long time.

With the introduction of the SQLOS in SQL Server 2005 a lot changed in how SQL Server processed requests, introducing schedulers, worker threads, and tasks. All the information for the various parts are stored in Dynamic Management Views (DMVs) or Dynamic Management Functions (DMFs), which are easily queried and return a lot of information about the internals of SQL Server.

Using these DMVs, we can view the progress of requests while they are being handled by a SQL Server scheduler and learn if they are waiting for any specific resources. The resources the requests are waiting for and the time they spend waiting for those resources are recorded as wait statistics, which is the main topic of this book.

CHAPTER 2

Querying SQL Server Wait Statistics

With the introduction of Dynamic Management Views (DMVs) in SQL Server 2005, viewing and analyzing wait statistics has become a lot easier and less tedious.

In SQL Server versions prior to SQL Server 2005, we were limited to the DBCC SQLPERF('WAITSSTATS') command to view wait statistics. Presently there are a variety of DMVs that return wait statistics-related information, and in this chapter, we will take a detailed look at four of the most useful DMVs: `sys.dm_os_wait_stats`, `sys.dm_os_waiting_tasks`, `sys.dm_exec_requests`, and `sys.dm_exec_session_wait_stats`.

Viewing wait statistics information is not only limited to DMVs though. We can also use the Windows Performance Monitor, or Perfmon, to view wait statistics information. SQL Server 2008 introduced yet another option to view wait statistics, Extended Events. While Extended Events were pretty complicated to work with in SQL Server 2008, meaning you would have to write an entire Extended Event session in T-SQL, Microsoft has drastically improved Extended Events in SQL Server 2012, making them a lot more user-friendly and easier to use.

SQL Server 2016 SP1 introduced two new methods to access wait statistics: through a new DMV called `sys.dm_exec_session_wait_stats` and by adding wait statistics information on a per-query basis inside execution plans.

In SQL Server 2017 Microsoft took recording wait statistics another step forward by including them inside the Query Store. The Query Store is a feature that was introduced in SQL Server 2016 and acts like a flightrecorder for your query workload, logging query statement, performance, and resource utilization.

We will take a look at all of the sources that capture wait statistics that are mentioned in the previous paragraphs inside this chapter, starting with the various DMVs. Because the Query Store feature has such a big impact on how you can troubleshoot and analyze query performance, including wait statistics in SQL Server 2017 and higher, we are going to take a thorough look at it in Chapter 3, “The Query Store.”

Sys.dm_os_wait_stats

The `sys.dm_os_wait_stats` DMV is probably one of the most important DMVs regarding wait statistics. This DMV is the replacement for the `DBCC SQLPERF('WAITSTATS')` command you would have had to use before SQL Server 2005. All of the information the `DBCC SQLPERF('WAITSTATS')` command returned is included in the `sys.dm_os_wait_stats` DMV, plus a little bit more.

The `sys.dm_os_wait_stats` DMV shows the total amount of wait time for every wait type since the start (or restart) of your SQL Server. It is also cumulative, adding wait time to the different wait types, resulting in an ever-increasing total. Querying the `sys.dm_os_wait_stats` DMV will give you insight into what your SQL Server has been waiting on the most since the time it started or was restarted. This can be helpful if you are looking for that grand total of wait time for every wait type, but many times you are interested in the wait time for a specific time segment. In this case it is possible to reset the `sys.dm_os_wait_stats` DMV without having to restart your SQL Server by using the `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` SQL command. This will reset all wait statistics information back to 0 again, meaning you will lose all information before the reset. In Chapter 4, “Building a Solid Baseline,” we will take a look at a method that does not completely reset the `sys.dm_os_wait_stats` DMV.

As with every DMV in SQL Server, we can run a query against the `sys.dm_os_wait_stats` DMV just like we would against a table, in this case `SELECT * FROM sys.dm_os_wait_stats`. The results of this query are shown in Figure 2-1.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	MISCELLANEOUS	0	0	0	0
2	LCK_M_SCH_S	0	0	0	0
3	LCK_M_SCH_M	0	0	0	0
4	LCK_M_S	8	1493	260	2
5	LCK_M_U	0	0	0	0
6	LCK_M_X	0	0	0	0
7	LCK_M_IS	0	0	0	0
8	LCK_M_IU	0	0	0	0
9	LCK_M_IX	0	0	0	0
10	LCK_M_SIU	0	0	0	0
11	LCK_M_SIX	0	0	0	0
12	LCK_M_UIX	0	0	0	0
13	LCK_M_BU	0	0	0	0

Figure 2-1. *Sys.dm_os_wait_stats*

Following are the available columns in the `sys.dm_os_wait_stats` DMV, along with a description of what each column can tell you:

- `wait_type` returns the wait type. The `sys.dm_os_wait_stats` will always return one row for every wait type possible in that specific SQL Server version.
- `waiting_tasks_count` shows a total of how many times a worker thread had to wait for that specific wait type.
- `wait_time_ms` returns the total wait time in milliseconds (1/1000 of a second) for that specific wait type since the start of the SQL Server instance or a manual reset of the DMV. This is the time a worker thread has spent in the Waiter List in the “SUSPENDED” state. It also includes the time the worker thread spent in the Runnable Queue in the “RUNNABLE” state while waiting for the scheduler to grant it processor time.
- `max_wait_time_ms` shows the maximum wait time in milliseconds a worker thread waited on that specific wait type.
- `signal_wait_time_ms` tells us the amount of time in milliseconds the worker thread spent in Runnable Queue waiting for processor time.

Signal wait times are unavoidable and normal in Online Transaction Processing (OLTP) systems where a large number of queries are being processed, all of them requesting time on the processor. The signal wait time is also an important metric for detecting CPU pressure. Generally speaking, as it depends on the hardware of your system, seeing signal wait time metrics higher than 25% of the total wait time can indicate CPU pressure, because the worker threads are waiting for the processor to become available instead of using resources.

You may have noticed that the `sys.dm_os_wait_stats` DMV does not return a column for the resource wait time. If we want to display the resource wait time as an additional column, we will need to calculate the value ourselves.

[Listing 2-1](#) shows a query you could use to analyze the `sys.dm_os_wait_stats` DMV. Besides the regular columns, it will add two more columns for every wait type returned, the resource wait time and the average wait time.

Listing 2-1. `sys.dm_os_wait_stats` with additional information

```
SELECT
    wait_type AS 'Wait Type',
    waiting_tasks_count AS 'Waiting Tasks Count',
    (wait_time_ms - signal_wait_time_ms) AS 'Resource Wait Time',
    signal_wait_time_ms AS 'Signal Wait Time',
    wait_time_ms AS 'Total Wait Time',
    COALESCE(wait_time_ms / NULLIF(waiting_tasks_count,0), 0) AS 'Average
    Wait Time'
FROM sys.dm_os_wait_stats;
```

This query will return results as shown in Figure [2-2](#).

	Wait Type	Waiting Tasks Count	Resource Wait Time	Signal Wait Time	Total Wait Time	Average Wait Time
49	ASYNC_IO_COM...	2	11	0	11	5
50	ASYNC_NETWO...	726	420	18	438	0
51	SLEEP_BPOOL_...	219	124	1	125	0
52	CHKPT	1	399	0	399	399
53	SLEEP_DBSTA...	5	391	0	391	78
54	SLEEP_MASTE...	0	0	0	0	0
55	SLEEP_MASTE...	2	39	0	39	19
56	SLEEP_MASTE...	1	386	2	388	388
57	SLEEP_TEMPD...	0	0	0	0	0
58	SLEEP_DCOMS...	1	19	0	19	19
59	SLEEP_TASK	318142	177929507	12723	177942230	559
60	SLEEP_SYSTE...	1	399	0	399	399
61	RESOURCE_SE...	0	0	0	0	0

Figure 2-2. *sys.dm_os_wait_stats expended with more wait information*

Having both the number of occurrences of a specific wait type and the total wait time makes it possible to calculate an average wait time (represented by the Average Wait Time column in Figure 2-2) for that specific wait type by dividing the `wait_time_ms` value by the `waiting_tasks_count` value.

The `sys.dm_os_wait_stats` is a powerful DMV with which you can retrieve a lot of information about the different wait types. It is also the basis of the wait statistics baseline methodology outlined in Chapter 4, “Building a Solid Baseline.”

Sys.dm_os_waiting_tasks

While the `sys.dm_os_wait_stats` DMV gives you the cumulative wait statistics information since server restart, the `sys.dm_os_waiting_tasks` DMV can give you information about what your SQL Server is currently waiting on. Querying this DMV will give you an overview of all the tasks that currently have worker threads waiting in the Waiter List or Runnable Queue for either resource or processor time.

Understanding sys.dm_os_waiting_tasks

Because the `sys.dm_os_waiting_tasks` DMV gives insight into what's waiting currently, it is usually the first DMV you query when using wait statistics to review the performance of your SQL Server instance. It also supplies some additional information for certain wait types that can be useful while troubleshooting.

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

Figure 2-3 shows the results of a `SELECT * FROM sys.dm_os_waiting_tasks;` query.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address
1	0x0000007F7D01C8C8	12	0	51066	XE_DISPATCHER_WAIT	NULL	NULL
2	0x0000007F7D01D468	NULL	NULL	70373146	XTP_PREEMPTIVE_TASK	NULL	NULL
3	0x0000007F7D01DC28	19	0	1550594	CHECKPOINT_QUEUE	0x0000008048ADEFA0	NULL
4	0x0000007F78B5C108	5	0	317	LAZYWRITER_SLEEP	NULL	NULL
5	0x0000007F78B5C4E8	7	0	70372313	KSOURCE_WAKEUP	NULL	NULL
6	0x0000007F78B5C8C8	8	0	863	SLEEP_TASK	NULL	NULL
7	0x0000007F78B5CCA8	17	0	171239	SP_SERVER_DIAGNOSTICS_SLEEP	0x0000000000000001	NULL
8	0x0000007F7453E8C8	29	0	70372699	HADR_NOTIFICATION_DEQUEUE	0x000000810DB4EF00	NULL
9	0x0000007F6E5428C8	32	0	70372323	BROKER_EVENTHANDLER	NULL	NULL
10	0x0000007F6E542CA8	9	0	863	BROKER_TO_FLUSH	NULL	NULL

Figure 2-3. sys.dm_os_waiting_tasks

Following is a list of columns returned by the `sys.dm_os_waiting_tasks` DMV and a description of the information they return:

- `waiting_task_address` shows the address of the task that is currently waiting.
- `session_id` gives us the ID of the session that is associated with the specific task.
- `exec_context_id` will return the ID of the execution context. This value will only change from the default of 0 if the task is being performed using parallelism. This means that the task is being executed using multiple threads instead of a single (serial) thread.
- `wait_duration_ms` shows us the time in milliseconds that the task has been waiting. Just like in the `sys.dm_os_wait_stats` DMV, this time includes both the resource wait time and the signal wait time.
- `wait_type` returns the wait type the task is currently waiting on.
- `resource_address` returns memory address information about the resource we are currently waiting for. Not all wait types will log this memory address, so it will frequently be returned as NULL.
- `blocking_task_address` will return the address of the task that is currently blocking the waiting task. When the task is not being blocked by another task, this column will return NULL.

- `blocking_session_id` returns the session ID of the session that is currently blocking the task. Just like the `blocking_task_address`, this information is only included when this task is being currently blocked by another task. It will return NULL when there is no blocking or when the session information about the blocking task cannot be retrieved or identified. We will explain blocking and locking in Chapter 8, “Lock-Related Wait Types,” when we discuss lock wait types.
- `blocking_exec_context_id` is another column dedicated to information regarding blocking. In this case it will return the ID of the execution context. This will only return a result other than NULL when a task gets executed using parallelism and one of the threads is responsible for the block. The `blocking_exec_context_id` can then be used to identify which one of the threads is responsible for the block.
- The last column of the DMV, `resource_description`, will give additional information about the resource the task is waiting for. There aren’t many wait types that will fill this column—most often parallelism, lock-, or latch-related wait types. It can be a very useful column, especially when analyzing lock- or latch-related wait types; in those cases we can pinpoint the database object (data page, row, table, etc.) whose availability we are waiting for. Some of the examples later in this book (most notably Chapter 8, “Lock-Related Wait Types,” and Chapter 9, “Latch-Related Wait Types”) will make use of this column to gather extra information about the resource we are waiting for.

Querying sys.dm_os_waiting_tasks

Because the `sys.dm_os_waiting_tasks` DMV returns a wealth of information, there are various ways to query it depending on what you want to analyze or troubleshoot.

One query that I see a lot on various forums on the Internet is the following:

```
SELECT * FROM sys.dm_os_waiting_tasks  
WHERE session_id > 50;
```

This query will filter out all of the SQL Server internal session IDs and only returns waiting tasks that originate from user sessions. The results of the query on my test SQL Server are shown in Figure 2-4.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address
1	0x0000007F7453C108	54	0	0	ASYNC_NETWORK_IO	NULL	NULL

Figure 2-4. *sys.dm_os_waiting_tasks where session_id is greater than 50*

While the method of filtering out internal SQL Server processes will work fine for many wait types and improves readability, there are specific wait types that will not be returned when running this query.

One good example of this is the THREADPOOL wait type, which we will discuss in Chapter 5, “CPU-Related Wait Types.” This wait type can have a large negative impact on the performance of your SQL Server but will not be returned if you only query the user sessions. This can impact your analysis because you are missing important facts.

Another reason to query the DMV without filtering on session IDs is that there is a big misconception about the relation of the session ID and whether or not the session ID is a user or internal session. While session IDs larger than 50 are generally considered to be user sessions, there is no guarantee that a session ID larger than 50 actually *is* a user session. It is possible that there is a need for SQL Server to have more than 50 internal sessions, in which case there is a chance you will see internal sessions with a session ID higher than 50 that you can mistake for a user session.

I believe the best way to query the sys.dm_os_waiting_tasks DMV is by selecting everything and only applying a filter if you are looking for a specific wait type or session. This will return many more rows than filtering on session IDs larger than 50, as you can see in Figure 2-5, but it will show you the complete picture and minimizes the chance that you might miss important wait types. A good idea might be to sort on the session_id column to make the results a little bit more readable without losing sight of the internal sessions.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address
3	0x00000007F7D01DC28	19	0	90266	CHECKPOINT_QUEUE	0x00000008048ADEFAD	NULL
4	0x00000007F78B5C108	5	0	906	LAZYWRITER_SLEEP	NULL	NULL
5	0x00000007F78B5C4E8	7	0	70567763	KSOURCE_WAKEUP	NULL	NULL
6	0x00000007F78B5C8C8	8	0	893	SLEEP_TASK	NULL	NULL
7	0x00000007F78B5CC48	17	0	66688	SP_SERVER_DIAGNOSTICS_SLEEP	0x00000000000000000001	NULL
8	0x00000007F7453E8C8	29	0	70568149	HADR_NOTIFICATION_DEQUEUE	0x0000000810DB4EF00	NULL
9	0x00000007F6E542C8C8	32	0	70567773	BROKER_EVENTHANDLER	NULL	NULL
10	0x00000007F6E542CA8	9	0	377	BROKER_TO_FLUSH	NULL	NULL
11	0x00000007F7D027088	10	0	1205	XE_TIMER_EVENT	NULL	NULL
12	0x00000007F7D027848	1	0	70568596	WAIT_XTP_HOST_WAIT	NULL	NULL
13	0x00000007F7D027C28	24	0	70568164	ONDemand_TASK_QUEUE	0x00000008046BDE8A0	NULL

Figure 2-5. sys.dm_os_waiting_tasks

Sys.dm_exec_requests

The `sys.dm_exec_requests` DMV returns information about all the requests that are currently getting processed by SQL Server.

Understanding sys.dm_exec_requests

Like the previous DMVs, we can query the `sys.dm_exec_requests` DMV with a simple `SELECT * FROM sys.dm_exec_requests;` to return everything that is currently executing. Figure 2-6 returns a small portion of the results on my test SQL Server.

	session_id	request_id	start_time	status	command	sql_handle	statement_start_offset
1	1	0	2019-01-22 10:15:38.357	background	XTP_CKPT_AGENT	NULL	NULL
2	2	0	2019-01-22 10:15:38.400	background	LOG WRITER	NULL	NULL
3	3	0	2019-01-22 10:15:38.417	background	RECOVERY WRITER	NULL	NULL
4	4	0	2019-01-22 10:15:38.417	background	LOCK MONITOR	NULL	NULL
5	5	0	2019-01-22 10:15:38.417	background	LAZY WRITER	NULL	NULL
6	6	0	2019-01-22 10:15:38.417	background	XIO_RETRY_WORKER	NULL	NULL
7	7	0	2019-01-22 10:15:39.210	background	SIGNAL HANDLER	NULL	NULL
8	8	0	2019-01-22 10:15:38.417	background	XIOLEASE_RENEWAL_WORKER	NULL	NULL
9	9	0	2019-01-22 10:15:39.167	background	BRKR TASK	NULL	NULL
10	10	0	2019-01-22 10:15:38.497	background	XE TIMER	NULL	NULL

Figure 2-6. sys.dm_exec_requests

The `sys.dm_exec_requests` DMV returns a lot more columns than the `sys.dm_os_wait_stats` or `sys.dm_os_waiting_tasks` DMVs we discussed earlier. To keep things readable I will only describe the columns we will frequently use for wait statistics analysis. Here is the list:

- `session_id` returns the ID of the session this request is associated with.
- `start_time` shows the date and time when the request got created. This can be a different time and even date than the time and date on which you are querying the DMV, especially when long-running queries are being executed.
- `command` returns information about what kind of action the request is performing. The most common commands are query related, like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, but there are many more commands depending on what is being executed by the request.
- `sql_handle` gives us a hash value of the SQL text that is being executed in the request. Not all requests have an SQL handle, and generally you should only see a SQL handle if the request was initiated by a user session and an SQL query is involved. The SQL handle hash can be used as input for the Dynamic Management Function (DMF) `sys.dm_exec_sql_text` to retrieve the query that is being executed by the request.
- `plan_handle` returns the hash value of the execution plan. An execution plan will show you the operations that were performed by SQL Server when executing the query and is a great source of query-execution information. We can use `plan_handle` the same way as the `sql_handle`, but instead of returning the query it will return the execution plan of the query. We can use the hash value as input for the DMF `sys.dm_exec_query_plan` in order to return the execution plan of the query that is being executed by the request.
- `wait_type` returns the current wait type if the request is either “SUSPENDED” or “RUNNABLE.” The value will be `NULL` if the request is currently being processed.

- The `last_wait_type` column returns the last wait type the request encountered if it had to wait during its execution.
- `total_elapsed_time` column returns the total time, in milliseconds, it took to process the request from the moment it got created.

There are many more columns available in this DMV that all have their different uses. A complete description is available on the Microsoft MSDN page at <https://msdn.microsoft.com/en-us/library/ms177648.aspx>, and I encourage you to go through the article. The `sys.dm_exec_requests` DMV is a great tool in your DBA toolkit and is one of those DMVs you will use frequently for all kinds of purposes besides analyzing wait statistics.

Querying `sys.dm_exec_requests`

The `sys.dm_exec_requests` DMV is one of the DMVs that can give us access to query statements and corresponding execution plans by returning the query and plan handles. If you are interested in this information, and most of the time you probably will be, you need to pass the `sql_handle` and `plan_handle` to their DMFs so the hashes turn into something we humans can read and understand.

Listing 2-2 shows a query against the `sys.dm_exec_requests` DMV and also retrieves the query statements and execution plans. I am excluding the session ID I am executing the query on and ignoring session IDs lower than 50 so as to keep the results small, and because I know for a fact that I am interested only in user queries for this example.

Listing 2-2. Query `sys.dm_exec_request` and include query statement and plan

```
SELECT
    r.session_id AS 'Session ID',
    r.start_time AS 'Request Start',
    r.[status] AS 'Current State',
    r.[command] AS 'Request Command',
    t.[text] AS 'Query',
    p.query_plan AS 'Execution Plan'
FROM sys.dm_exec_requests r
OUTER APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
```

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

```
OUTER APPLY sys.dm_exec_query_plan(r.plan_handle) p  
WHERE r.session_id > 50 AND r.session_id <> @@SPID;
```

On my test system I got the results as shown in Figure 2-7.

	Session ID	Request Start	Current State	Request Command	Query	Execution Plan
1	54	2019-01-24 20:03:32.173	suspended	SELECT	select * from Sales.Salesorderdetail sod inner ...	ShowPlanXML xmlns="http://schemas.microsoft.com..."/>

Figure 2-7. Results of Listing 2-2

Using the query in Listing 2-2, we can immediately see that session ID 55 is performing a SELECT query, as the query column shows the complete statement that is being executed. The Execution Plan column returns the execution plan in an XML format. The great part of the SQL Server Management Studio is that we can click the XML link that is returned to view the graphical execution plan, as shown in Figure 2-8.



Figure 2-8. Execution plan

Using the execution plan we can get some insight into how the query is getting executed by the SQL Server engine. We won't get into the details about execution plans in this book, but you will be using them frequently when you are optimizing query performance, so it is good to know how you can access them from the sys.dm_exec_requests DMV. A good place to start if you want to learn more about execution plans is Grant Fritchey's *Execution Plan Basics* at www.simple-talk.com/sql/performance/execution-plan-basics/.

Sys.dm_exec_session_wait_stats

One of the latest additions to the wait statistic-related DMVs is the **sys.dm_exec_session_wait_stats** DMV. It was introduced in SQL Server 2016 SP1 and returns wait statistics information on a per session level. If you remembered reading Chapter 1,

“Wait Statistics Internals,” of this book, a session is an active connection a user or process has with SQL Server. A session can have multiple requests, which in turn can have multiple tasks performing the actions required to execute a query.

Figure 2-9 shows the columns of the DMV as well as some wait statistics information from my test system.

	session_id	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	51	PAGELATCH_SH	1	0	0	0
2	51	PAGELATCH_EX	1	0	0	0
3	51	PAGEIOLATCH_SH	14	8	1	0
4	51	SOS_SCHEDULER_YIELD	2	0	0	0
5	51	MSQL_XP	2	1	1	0
6	51	PREEMPTIVE_OS_CRYPTOPS	1	0	0	0
7	51	PREEMPTIVE_OS_CRYPTACQUIRECONTEXT	4	0	0	0
8	51	PREEMPTIVE_OS_GETPROCADDRESS	2	0	0	0
9	51	PREEMPTIVE_OS_LOADLIBRARY	1	2	2	0
10	51	PREEMPTIVE_OS_REPORTEVENT	2	0	0	0

Figure 2-9. sys.dm_exec_session_wait_stats

Does the preceding figure look familiar? It probably does since it is practically identical to the **sys.dm_os_wait_stats** DMV but with an additional column for the **session_id**.

What is important to point out is that wait statistics information that is recorded through this DMV is cumulative for all the actions that a specific session performed while it was active. For instance, if you execute ten different queries inside a single session, all the wait times that are returned by the DMV are the total wait time for those ten queries. This means it is very important to understand what happened during the lifetime of a session. Has the session already been busy executing large batches of queries? Or has it only executed a single query statement? Knowing the answers to these questions is very important before resorting to this DMV to analyze session wait statistics.

Also, session IDs are reused after a session has been closed, meaning that if you are not careful you can see the wait statistics for a new session that is reusing the session ID. When a session ID gets reused (or reset when using connection pooling), all the wait statistics information inside this DMV for that specific session is reset as well.

With the preceding information in mind, we can conclude this DMV is not directly useful as a “first-place-to-look” when performance is reported to be slow. The DMV still has its place though, especially if you can reproduce a specific performance issue with a

specific action where multiple queries are involved. In that situation you can zoom in on a specific session ID and reproduce the issue, capturing wait statistics for everything that happens during the execution of the queries.

Combining DMVs to Detect Waits Right Now

Now that we have taken a look at some of the most important DMVs for wait statistics analysis, let's go into an example of how you could use these DMVs to find out what is slowing down your SQL Server. Gathering this information will not solve your problems immediately, but it will give you a clue as to where to start looking for a solution.

Consider the following scenario: You are the database administrator (DBA) for a large company that uses a single database to store all its sales information. The database is running on a SQL Server 2014 instance, and every day a few hundred users query the database.

Normally everything is running fine—users can access the information they want quickly, and everyone who needs to work with the database is happy. Today, however, is not a good day for you as the DBA. The phone hasn't stopped ringing since 10 AM and some users are gathering at the door of your office with an angry look in their eyes—querying and inserting sales information is incredibly slow.

Since this book is about wait statistics, let's take a look at how we could analyze wait statistics information about the performance problem in the scenario.

We know the `sys.dm_os_wait_stats` DMV shows cumulative wait statistics information, so for this scenario it wouldn't be much help. A much better starting place would be the `sys.dm_os_waiting_tasks` DMV, since it will show us all the tasks that are waiting right now.

We run the following query against the `sys.dm_os_waiting_tasks` DMV:

```
SELECT * FROM sys.dm_os_waiting_tasks  
ORDER BY session_id ASC;
```

While scrolling down to the bottom of the results, we see a number of user sessions with waiting tasks, as shown in Figure 2-10.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address
16	0x00000007F7D01C8C8	12	0	2928	XE_DISPATCHER_WAIT	NULL	NULL
17	0x00000007F75A024E8	16	0	1851	SQLTRACE_INCREMENTAL_FLUSH_SLEEP	NULL	NULL
18	0x00000007F78B5CCA8	17	0	65818	SP_SERVER_DIAGNOSTICS_SLEEP	0x0000000000000001	NULL
19	0x00000007F7D01DC28	19	0	24115	CHECKPOINT_QUEUE	0x000000804ADEFA0	NULL
20	0x00000007F7D027C28	24	0	70867309	ONDemand_TASK_QUEUE	0x0000008046BDE8A0	NULL
21	0x00000007F7453E8C8	29	0	70867294	HADR_NOTIFICATION_DEQUEUE	0x000000810DB4EF00	NULL
22	0x00000007F6E542C8	32	0	70866918	BROKER_EVENTHANDLER	NULL	NULL
23	0x00000007F72FA88C8	33	0	365	HADR_FILESTREAM_IOMGR_IOCOMPLE...	NULL	NULL
24	0x00000007F72FA8108	42	0	70866946	BROKER_TRANSMITTER	NULL	NULL
25	0x00000007F72FA84E8	43	0	70866946	BROKER_TRANSMITTER	NULL	NULL
26	0x00000007F7453ECA8	52	0	15988	LCK_M_IS	0x0000007F78A8B540	NULL
27	0x00000007F7453C4E8	56	0	9308	LCK_M_IS	0x0000007F5897F980	0x00000007F7453ECA8
28	0x00000007F7453D848	57	0	11157	LCK_M_IS	0x0000007F5897F9C0	0x00000007F7453ECA8

Figure 2-10. Results of a query against the sys.dm_os_waiting_tasks DMV

We notice that the wait times for sessions 52, 56, 57 are pretty high, and we also notice that they are all waiting with a wait type LCK_M_S. Without going into too many details about this particular wait type, it will be discussed in detail in Chapter 8, “Lock-Related Wait Types,” it is enough to know that this wait type is related to locking. Apparently, those sessions are waiting to place a lock, which means they are probably being blocked by another process that has a lock on the same object. We can extract locking and blocking information from the sys.dm_os_waiting_tasks DMF by looking at the blocking_ columns. For readability reasons I modified the preceding query to return only blocking information from the sys.dm_os_waiting_tasks DMV. Figure 2-11 shows those columns.

	session_id	wait_type	blocking_task_address	blocking_session_id	blocking_exec_context_id
21	29	HADR_NOTIFICATION_DEQUEUE	NULL	NULL	NULL
22	32	BROKER_EVENTHANDLER	NULL	NULL	NULL
23	33	HADR_FILESTREAM_IOMGR_IOCOMPLE...	NULL	NULL	NULL
24	42	BROKER_TRANSMITTER	NULL	NULL	NULL
25	43	BROKER_TRANSMITTER	NULL	NULL	NULL
26	52	LCK_M_IS	NULL	54	NULL
27	56	LCK_M_IS	0x0000007F7453ECA8	52	NULL
28	57	LCK_M_IS	0x0000007F7453ECA8	52	NULL

Figure 2-11. Blocking information from the sys.dm_os_waiting_tasks DMV

From what we can see here, sessions 56 and 57 are being blocked by session 52. Session 52, however, is being blocked by session ID 54. We don’t see this session ID returned in the sys.dm_os_waiting_tasks DMV, which means the session is currently executing and isn’t waiting on any resources.

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

Let's check another DMV, `sys.dm_exec_requests`, to get some information about session 54:

```
SELECT * FROM sys.dm_exec_requests  
WHERE session_id = 54;
```

Figure 2-12 shows the results of this query.

session_id	request_id	start_time	status	command	sql_handle	statement_start_offset	statement_end_offset	plan...
------------	------------	------------	--------	---------	------------	------------------------	----------------------	---------

Figure 2-12. Results of a query against `sys.dm_exec_requests`

Remember when I wrote the `sys.dm_exec_requests` DMV return information about requests currently being processed? Apparently, session ID 54 doesn't have an outstanding request since no information is being returned.

If we want to find out more information about this session, we can use the `sys.dm_exec_sessions` DMV we discussed in Chapter 1, "Wait Statistics Internals," by executing the following query:

```
SELECT  
    session_id,  
    [status],  
    [host_name],  
    [program_name],  
    login_name,  
    is_user_process,  
    open_transaction_count  
FROM sys.dm_exec_sessions  
WHERE session_id = 54;
```

This query returns the results shown in Figure 2-13.

session_id	status	host_name	program_name	login_name	is_user_process	open_transaction_count
1	54	sleeping	EVDL-SQL2017-01	Microsoft SQL Server Management Studio - Query	EVDL-SQL2017-01\Administrator	1

Figure 2-13. Results from `sys.dm_exec_sessions`

We can assume that session ID doesn't currently have any running requests since its status is "sleeping," which is why the query against the `sys.dm_exec_requests` didn't return any information. If we look at the `program_name` column, we can see that this session was initiated from the Microsoft SQL Server Management Studio program by the `EVDL-SQL2017-01\Administrator` user.

I included the `is_user_process` column to make sure it is a user session, and the `open_transaction_count` column shows us that this user session has an open transaction.

We now know enough information to take corrective actions. We know who the user is who is blocking our other tasks, and we can decide to give him a call about what he is currently performing, or we can choose to end his session. Ending a user session by using the `KILL [session_id]` command should always be your last resort because we could be interrupting something important. Ending a session with the `KILL` command will result in a rollback of the running transaction, undoing all the changes it performed, which can take a long time to complete. In this case, I accept the risk of a rollback and will end the session myself:

```
KILL 54;
```

Immediately after we kill session ID 54 users report that their queries are running again. If we query the `sys.dm_os_waiting_tasks` DMV to give us information about those session IDs nothing gets returned, meaning they are no longer being blocked.

Hopefully this example has given you insight into how you can use the various DMVs available in SQL Server to gather information about tasks that are currently waiting. In this case the example consisted of a transaction that was blocking other queries, and we decided to kill the user session that was responsible for the blocking lock. In many situations the solution isn't this relatively simple, but the method of gathering wait statistics information to drill down to the bottom of the problem can be used in almost every performance-related incident.

As I noted in the beginning of this section, just looking at the wait statistics alone will not, in most cases, solve a performance problem, but it is a good starting point to begin your investigation. To get a complete picture about the performance of your system, we will often combine the wait statistics information with other metrics, like those from the Windows Performance Monitor, other DMVs, or vendor-specific information (like storage metrics).

Figure 2-14 shows a flowchart of how you could use wait statistics information to analyze a performance problem.

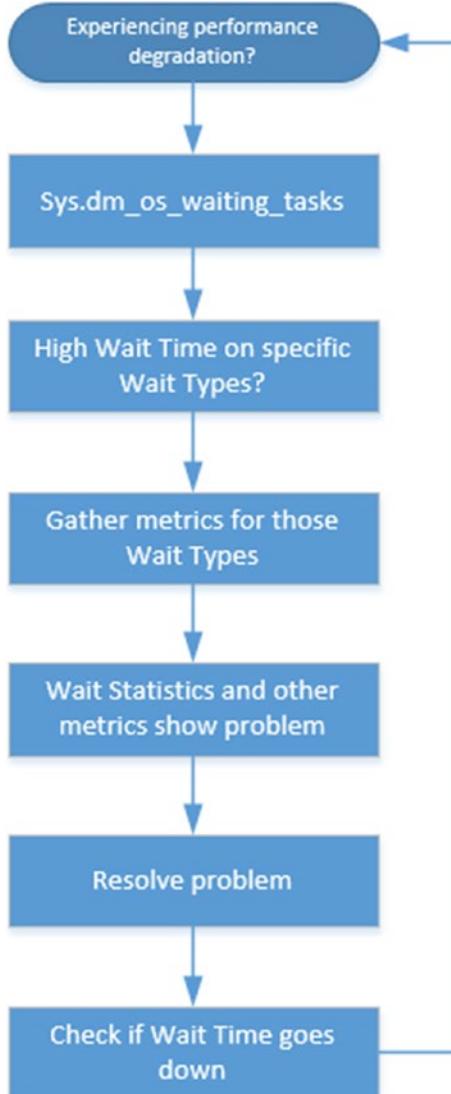


Figure 2-14. Wait statistics flowchart

We will expand upon this flowchart in Chapter 4, “Building a Solid Baseline,” where we will introduce baselines to the wait statistics analysis method.

Viewing Wait Statistics Using Perfmon

One of the most important tools for accessing the extra metrics we need when analyzing wait statistics is the Windows Performance Monitor, or Perfmon. Perfmon is available on every Windows Operating System and contains counters for just about every part of the system, including SQL Server-related performance counters. You can start Perfmon by executing the **perfmon** command from either a Windows Run dialog or the command line.

In addition to giving us information about the performance of our system, Perfmon can also be used to view wait statistics. You can view these counters under the **SQLServer:Wait Statistics** category when adding counters inside the Perfmon application, as shown in Figure 2-15.

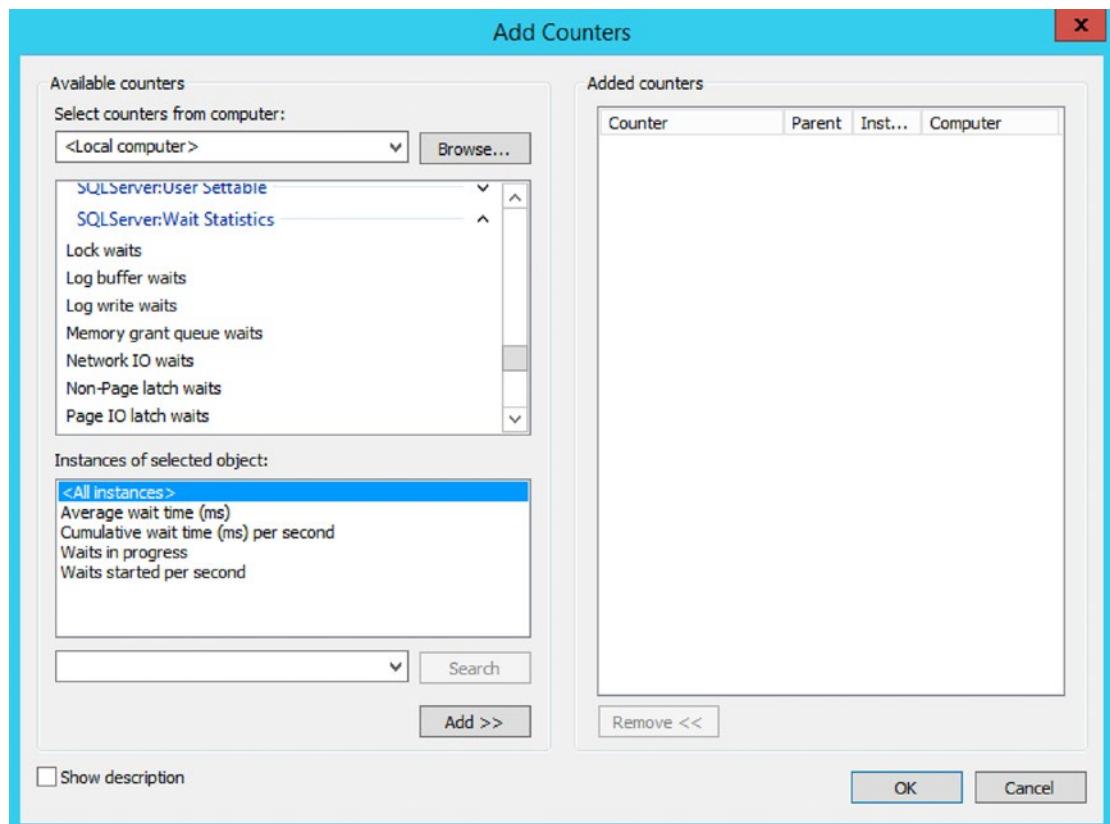


Figure 2-15. Wait statistics counters inside Perfmon

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

One thing you'll notice in Figure 2-15 is that the wait statistics inside Perfmon are grouped inside categories. We won't find information about specific wait types here, so if we want to use Perfmon to analyze wait statistics we should have a general idea of what category a specific wait type belongs to in Perfmon. It is able to display an average wait time, cumulative wait time, the current total number of waits, and the amount of new waits started for every wait statistics category. If we are interested in a higher-level view—for instance, we want to know how many tasks are waiting for lock-related wait types—we can use Perfmon to give us that information. If we want to have more detail about specific wait type information, we should use the sys.dm_os_wait_stats or sys.dm_os_waiting_tasks DMVs we discussed earlier.

One nice feature of Perfmon is that it can convert the measurements directly into graphs, giving us a more visual way to look at the information without having to create the graphs ourselves. Figure 2-16 is an example of a graph where we are showing the “Average wait time” and “Waits started per second” for the Lock waits category.

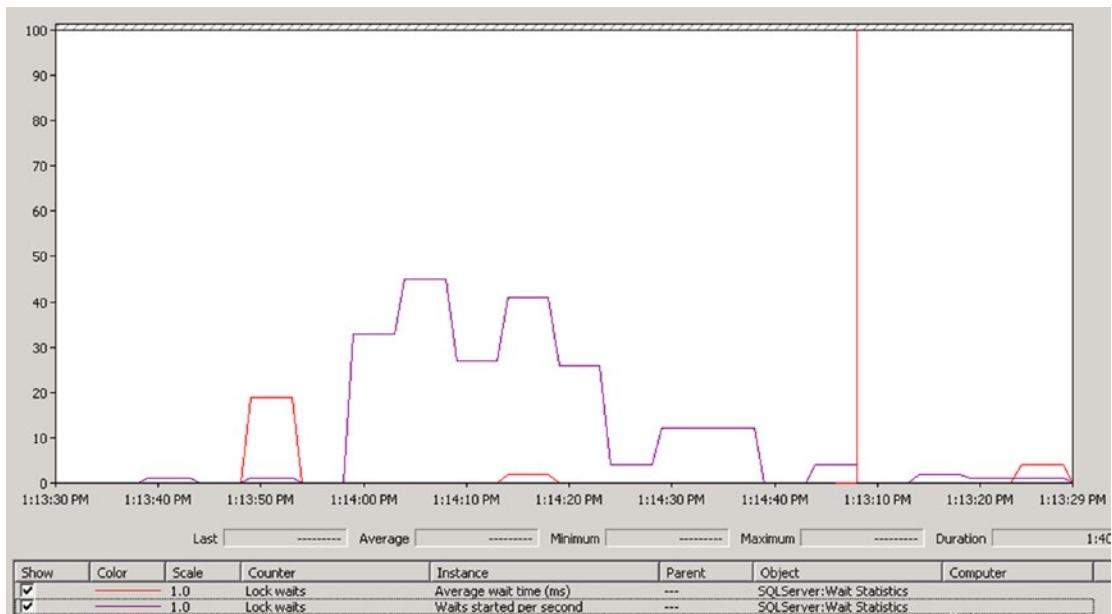


Figure 2-16. Perfmon graph showing wait statistics information

During this book we will use Perfmon a lot for analyzing metrics related to specific wait types, like CPU time, disk latency, and memory usage. We will not make much use of the wait statistics counters inside Perfmon, because I believe the SQL Server DMVs are better suited for this since they supply the level of detail needed for a complete analysis.

Capturing Wait Statistics Using Extended Events

Most of the wait statistics information in SQL Server is recorded cumulatively, and because so many internal processes also generate wait statistics it can be difficult to detect what impact a single query has on them. This is where Extended Events come in; with Extended Events it is possible to capture the exact wait times a query encountered and on what wait types it had to wait. This information can help us analyze those queries that have a large impact on our system and possibly optimize them so their impact becomes smaller. Or we could capture queries that encounter a specific wait type while executing.

Extended Events were introduced in SQL Server 2008 and are, more or less, a replacement for the SQL Server Profiler. Microsoft has announced the deprecation of the SQL Server Profiler and advises us to move to Extended Events. Extended Events are much more powerful than the SQL Server Profiler, and the number of events we can capture with Extended Events keeps growing with every release of SQL Server, while the number of events in the SQL Server Profiler remains the same. Also performance is a good reason to use Extended Events. Capturing Extended Events is more lightweight than using the SQL Server Profiler.

Extended Events have a reputation of being difficult to work with, and, while this was especially true in SQL Server 2008 when they were first introduced, working with Extended Events became a lot easier in SQL Server 2012 when it became possible to create Extended Event sessions using the GUI.

There are many different wait-related events available when working with Extended Events. We can view these events by running a query against the `sys.dm_xe_map_values` DMV, which holds all the different Extended Events event types:

```
SELECT *
FROM sys.dm_xe_map_values
WHERE name = 'wait_types';
```

Figure 2-17 shows a small part of the results of this query.

	name	object_package_guid	map_key	map_value
10	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	1173	ASSEMBLY_FILTER_HASHTABLE
11	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	358	ASSEMBLY_LOAD
12	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	115	ASYNC_DISKPOOL_LOCK
13	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	98	ASYNC_IO_COMPLETION
14	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	923	ASYNC_OP_COMPLETION
15	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	921	ASYNC_OP_CONTEXT_READ
16	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	922	ASYNC_OP_CONTEXT_WRITE
17	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	313	ASYNC_TRANSPORT_CONNECTION_DISPATCH
18	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	311	ASYNC_TRANSPORT_DISPATCH
19	wait_types	BD97CC63-3F38-4922-AA93-607BD12E78B2	317	ASYNC_TRANSPORT_DISPATCH_MUTEX

Figure 2-17. Results of sys.dm_xe_map_values

In total there are about 1260 different wait statistics-related events available in SQL Server 2019 CTP2.4. These events do not map one-on-one against the different wait types, and, as a matter of fact, in some cases the names of the wait types do not match those of the events, even though they have the same meaning. An example of this is the ASYNC_NETWORK_IO wait type, which is named NETWORK_IO by Extended Events. Jonathan Kehayias wrote a blog post at [SQLskills.com](https://www.sqlskills.com/blogs/jonathan/mapping-wait-types-in-dm_os_wait_stats-to-extended-events/) mapping some of the wait types to Extended Events; you can take a look at it here: www.sqlskills.com/blogs/jonathan/mapping-wait-types-in-dm_os_wait_stats-to-extended-events/.

While we won't go into details about Extended Events in this book, I would like to show you how you can use them to capture wait statistics-related information using the Extended Events GUI and T-SQL.

Capture Wait Statistics Information for a Specific Query

Let's take a look at how we can configure an Extended Event session to capture wait statistics information for a specific query. We will set a filter on a session ID that will execute the query, then execute the query we want to analyze.

The first thing we are going to do is open up the SQL Server Management Studio and connect to a SQL Server instance. Keep in mind that a GUI for Extended Events was added in SQL Server 2012, so if you plan on following the steps here you will need a SQL Server 2012 or higher SQL Server instance.

Once we are connected, we open the Management folder and then choose the Extended Events option. We right-click the Sessions folder and select the option New Session, as shown in Figure 2-18.

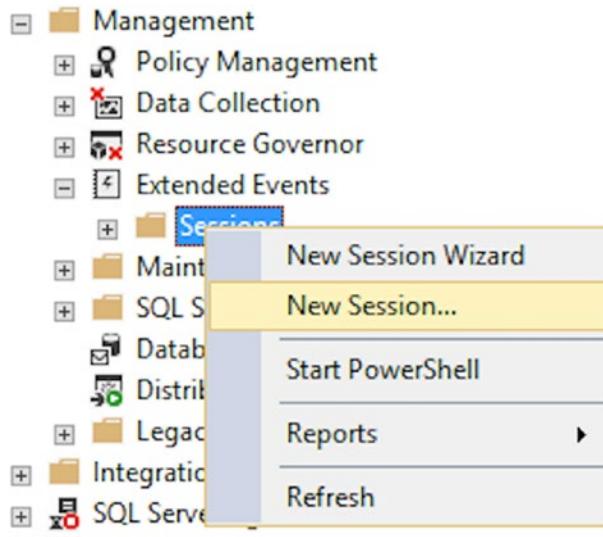


Figure 2-18. Adding new Extended Event session

The New Session dialog will appear where we can enter a name for this Extended Event session and set some additional options. We will ignore those options for now, and just fill in the name of the Extended Event session, as shown in Figure 2-19.

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

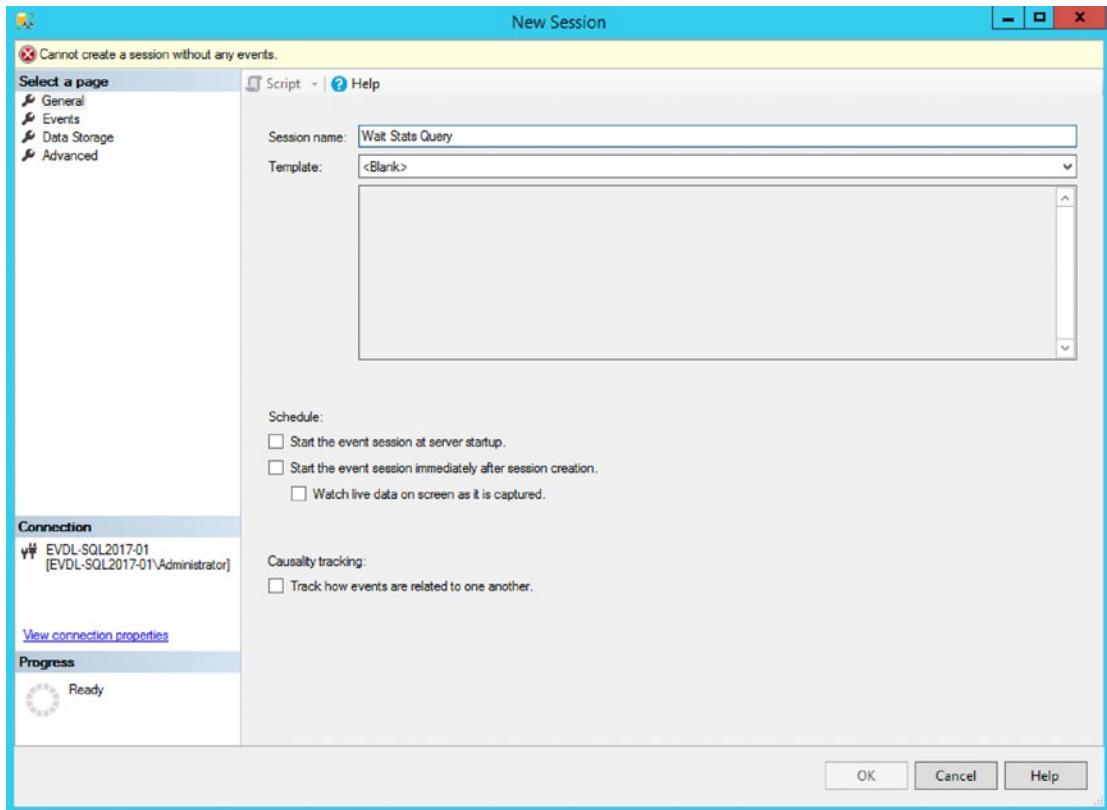


Figure 2-19. Configuring wait statistics Extended Event session

The next step is configuring which events this Extended Event session needs to monitor, which we can do by selecting the Events page in the New Session dialog.

Since we are interested in wait statistics information, I searched for the `wait_info` event in the Events Library and added it to the Selected Events box, as shown in Figure 2-20.

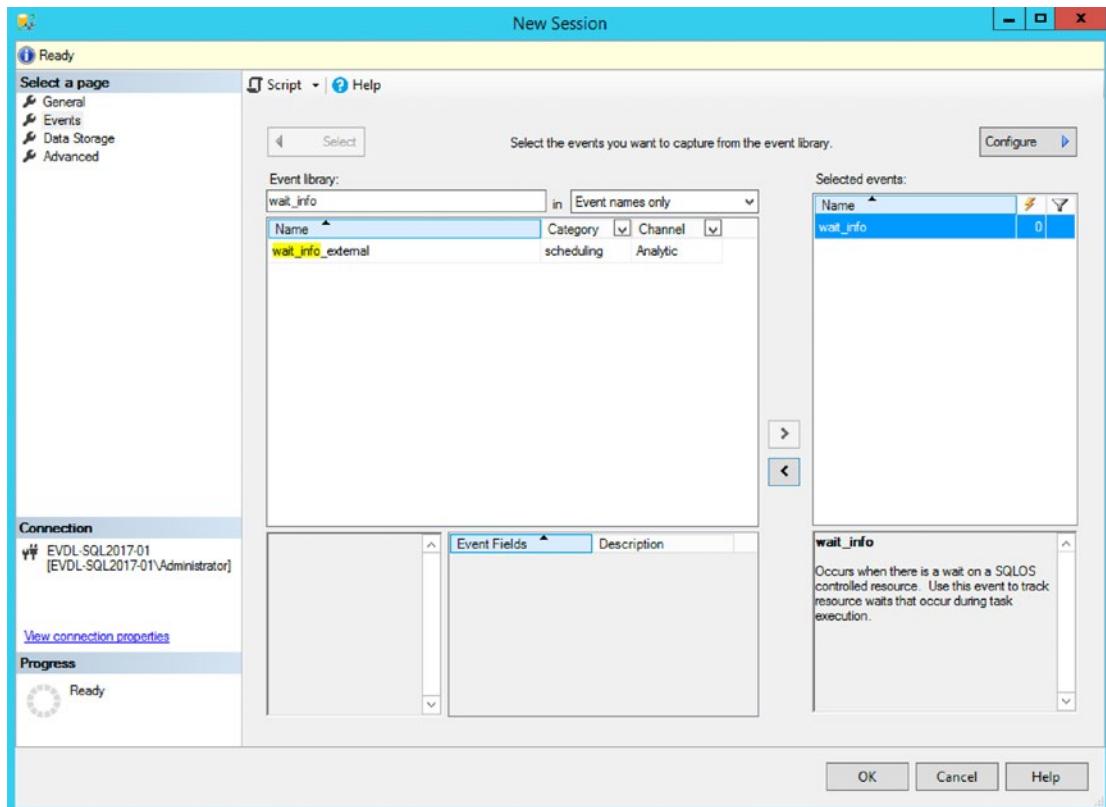


Figure 2-20. Selecting an event to monitor

If we were to save this Extended Event session now, we would capture information for every task that has to wait for a resource. Since we are interested in the wait statistics associated with a specific query, we will configure a filter to only return wait statistics information for a specific session. To do this we can click the Configure button in the New Session dialog, which will open a new section where we can select Global Fields, which will record extra information when a `wait_info` event is triggered. In this case, I checked the `sql_text` global field, as shown in Figure 2-21, so we can view the actual query when an event is captured.

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

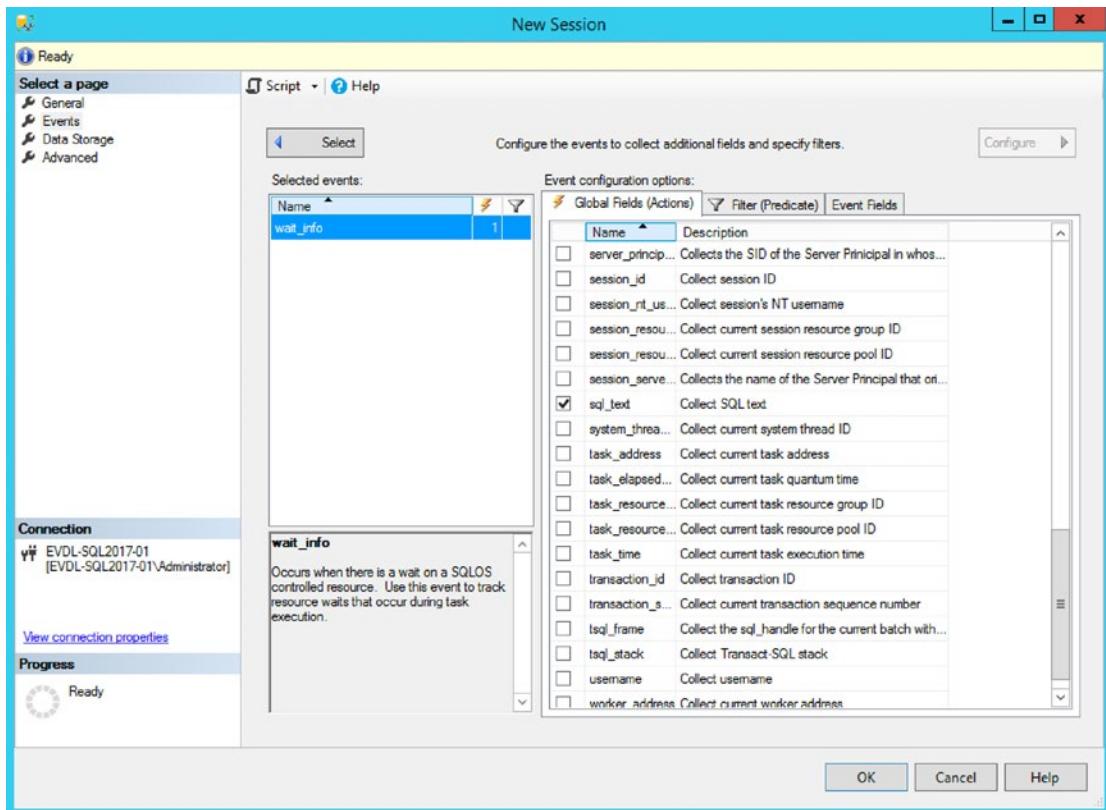


Figure 2-21. Setting the `sql_text` global field

Next up is the Filter (Predicate) tab. Here we will set a filter that will only capture events from a specific session ID. We can do this by clicking inside the Field box and selecting the `sqlserver.session_id` field, then setting the Value to the session ID we want to monitor. In this case I configured the filter to only capture events for session ID 52, as shown in Figure 2-22.

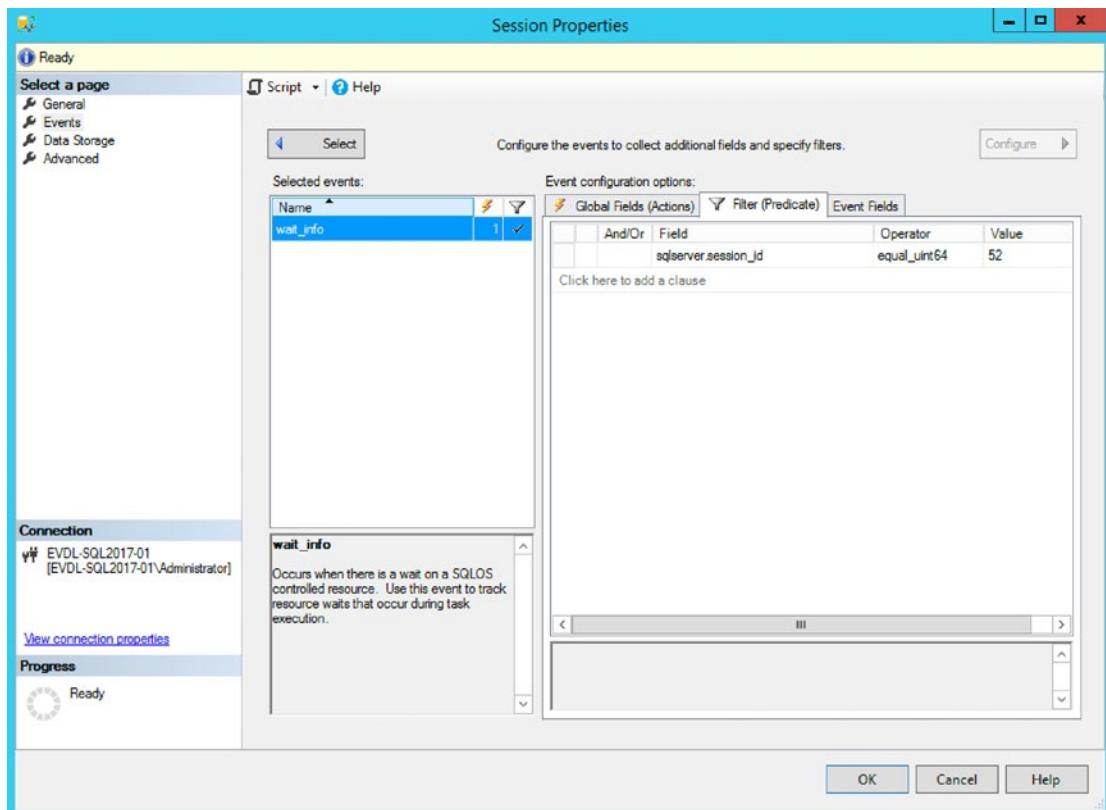


Figure 2-22. Setting an event filter

That's all we need to configure for now, so we can click OK to close this dialog and save the Extended Event session.

By default the Extended Event session will not be automatically started after it is created. To do this we have to open up the Sessions folder again by navigating to the Management ► Extended Events folder in SQL Server Management Studio. We right-click the Extended Event session we just created and select the Start Session option, as shown in Figure 2-23.

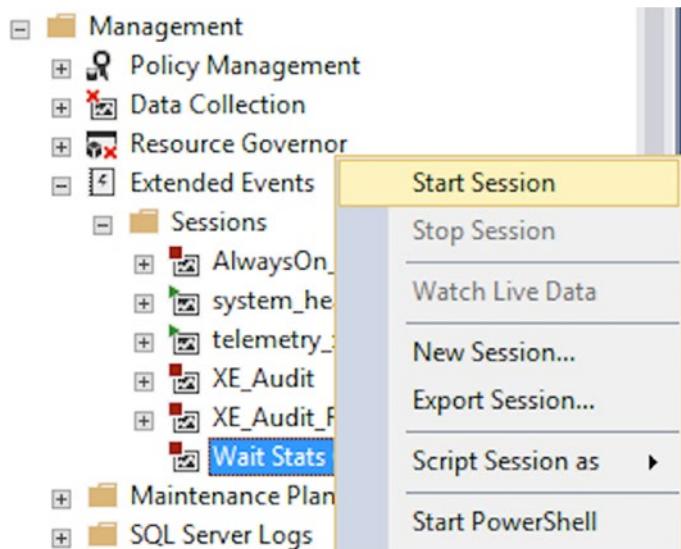


Figure 2-23. Start Extended Event session

After we have started the Extended Events session it will begin collecting information. We can view this information as it gets gathered by selecting the Watch Live Data option. This will open up a new tab in SQL Server Management Studio where we can watch the Extended Event session. Viewing live Extended Event data takes a little overhead, but this is far lower than the overhead of using SQL Profiler. If you are worried about the overhead of viewing live data, you could choose to write the Extended Event session to an event file by adding a file location inside the Data Storage page of your Extended Event session.

For this example I executed a simple query against the AdventureWorks database to return everything in the Person.Person table, as follows:

```
SELECT *
FROM Person.Person;
```

I pay close attention when setting the filter in the Extended Event session to match the session ID of the tab in SQL Server Management Studio where I am executing the query. The session ID can be found when looking at the number between parentheses on the tab. The Extended Events Live Data tab returned the information shown in Figure 2-24.

	name	timestamp
▶	wait_info	2019-01-24 20:29:43.2561393
	wait_info	2019-01-24 20:29:43.2562078
	wait_info	2019-01-24 20:29:43.2562207

Event: wait_info (2019-01-24 20:29:43.2561393)

Details	
Field	Value
duration	0
opcode	Begin
signal_duration	0
sql_text	select * from person.person
wait_resource	0
wait_type	NETWORK_IO

Figure 2-24. Live wait statistics information from an Extended Event session

As you can see in Figure 2-24, our request encountered a NETWORK_IO wait type. This is one of those examples where the wait name in Extended Events doesn't match the one in the wait statistics DMVs. The NETWORK_IO wait name is the same wait as the ASYNC_NETWORK_IO wait type the SQLOS uses. We can view the query we executed in the sql_text field.

There are many more global fields you can include in the Extended Events session that might be interesting to capture, like the Execution Plan handle or the Task Execution Time. All of these global fields will give you additional information that is shown when Extended Event session information is returned, giving you an impressive amount of detail.

If, for some reason, you do not want to use the GUI to create and run an Extended Event session or you are running SQL Server 2008, you can use T-SQL to create and configure one. To create the same Extended Event session as we did using the GUI, you can execute the query seen in Listing 2-3.

Listing 2-3. Create wait statistics Extended Event session

```
CREATE EVENT SESSION [WaitStats Query] ON SERVER  
  
ADD EVENT sqlos.wait_info  
(  
    ACTION(sqlserver.sql_text)  
    WHERE ([sqlserver].[session_id]=(52))  
)  
  
ADD TARGET package0.event_file  
(  
    SET filename = N'E:\Data\WaitStats_XE.xel', metadatafile = N'E:\Data\  
    WaitStats_XE.xem'  
)
```

We included the metadata file in the preceding script by setting the metadatafile parameter. If you are running SQL Server 2012 or higher, this is no longer required.

The easiest way to log the Extended Event session is by saving it to a file; in this case my filename is E:\Data\WaitStats_XE.xel (SQL Server will add a unique numeric identifier to the filename, in this case the actual filename is WaitStats_XE_0_130702270937280000.xel). I also included the filter on session ID 52 to capture wait statistics generated by that session.

The next thing we want to do is start the Extended Event session, which we can do by executing the ALTER EVENT SESSION command:

```
ALTER EVENT SESSION "WaitStats Query" ON SERVER STATE = start;
```

We then execute the same query as we did in the Extended Events GUI example under the session ID we are filtering on. After letting the Extended Event session run for a little while, we can stop it using the ALTER EVENT SESSION command:

```
ALTER EVENT SESSION "WaitStats Query" ON SERVER STATE = stop;
```

Now that we have stopped the Extended Event session, we need to import the information in the file (as XML) into a table so we can actually see what the session captured; we do this using the `sys.fn_xe_file_target_read_file` function. We can then parse the XML information to return the results in a more readable format. The query in Listing 2-4 can be used to read an Extended Events file, import it into a temporary table, and return the results as rows.

Listing 2-4. Return Extend Event file as rows

```
-- Check if temp table is present
-- Drop if exist
IF OBJECT_ID('tempdb..#XE_Data') IS NOT NULL
DROP TABLE #XE_Data

-- Create temp table to hold raw XE data
CREATE TABLE #XE_Data
(
    XE_Data XML
);
GO

-- Write contents of the XE file
-- into our table
INSERT INTO #XE_Data
(
    XE_Data
)
SELECT
    CAST (event_data AS XML)
FROM sys.fn_xe_file_target_read_file
(
    'E:\Data\WaitStats_XE_0_130702270937280000.xel',
    'E:\Data\WaitStats_XE_0_130702270940210000.xem',
    null,
    null
);
GO
```

CHAPTER 2 QUERYING SQL SERVER WAIT STATISTICS

```
-- Query information from our temp table
SELECT
    XE_Data.value ('(/event/@timestamp)[1]', 'DATETIME') AS 'Date/Time',
    XE_Data.value ('(/event/data[@name="opcode"]/text)[1]', 'VARCHAR(100)')
    AS 'Operation',
    XE_Data.value ('(/event/data[@name="wait_type"]/text)[1]',
    'VARCHAR(100)') AS 'Wait Type',
    XE_Data.value ('(/event/data[@name="duration"]/value)[1]', 'BIGINT') AS
    'Wait Time',
    XE_Data.value ('(/event/data[@name="signal_duration"]/value)[1]',
    'BIGINT') AS 'Signal Wait Time',
    XE_Data.value ('(/event/action[@name="sql_text"]/value)[1]',
    'VARCHAR(100)') AS 'Query'
FROM #XE_Data
ORDER BY 'Date/Time' ASC
;
```

The result of the query in Listing 2-4 can be seen in Figure 2-25.

	Date/Time	Operation	Wait Type	Wait Time	Signal Wait Time	Query
13	2019-01-24 19:34:28.687	Begin	SOS_SCHEDULER_YIELD	0	0	select * from person.person
14	2019-01-24 19:34:28.687	End	SOS_SCHEDULER_YIELD	0	0	select * from person.person
15	2019-01-24 19:34:28.693	Begin	SOS_SCHEDULER_YIELD	0	0	select * from person.person
16	2019-01-24 19:34:28.693	End	SOS_SCHEDULER_YIELD	0	0	select * from person.person
17	2019-01-24 19:34:28.693	Begin	NETWORK_IO	0	0	select * from person.person
18	2019-01-24 19:34:28.697	End	NETWORK_IO	3	0	select * from person.person
19	2019-01-24 19:34:28.697	Begin	NETWORK_IO	0	0	select * from person.person
20	2019-01-24 19:34:28.697	End	NETWORK_IO	0	0	select * from person.person
21	2019-01-24 19:34:28.700	Begin	NETWORK_IO	0	0	select * from person.person
22	2019-01-24 19:34:28.707	End	NETWORK_IO	6	0	select * from person.person

Figure 2-25. Results of the query in Listing 2-4

Most of the columns speak for themselves in terms of the row data they return. Two columns that deserve some extra explanation are the Operation and Wait Time columns. The Operation column will show you the beginning or the end of the wait event. The Wait Time column will return the wait time in milliseconds, but it will only be recorded at the end of an operation.

Analyzing Wait Statistics on a Per-Query Basis Using Execution Plans

So far we have mostly looked at aggregated wait times that were generated by either various background processes or by the queries we executed. Since I was the only one that was executing queries against my test machine, it is very easy to correlate wait times to my specific queries. Unfortunately, on busy systems where many queries are constantly being executed by a large number of sessions, the various wait statistics DMVs are almost useless to analyze wait types and wait times for specific queries.

Thankfully, the release of SQL Server 2016 SP1 changed that scenario and introduced wait statistics capture inside query execution plans! This means you can easily see what wait types and wait times your query encountered while running. Even though it might seem obvious, this means that per-query wait statistics are only available when looking at the actual execution plan, not the estimated execution plan.

The actual execution plan is the execution plan that was used during the execution of the query. There is an option in SQL Server Management Studio to look at the estimated execution plan. When used, the SQL Server engine compiles the execution plan which is most likely to be used during the query's execution; however, it does not actually execute the query itself. Since there is no query execution, there are also no wait statistics to record while compiling the estimated execution plan.

The easiest way to expose per-query wait statistics is by enabling the **Include Actual Execution Plan** option, shown in Figure 2-26, by clicking the “Query – Include Actual Execution Plan” menu item or by using the key combination CTRL_M, and then executing your query.



Figure 2-26. Include Actual Execution Plan option

When you execute your query with the Include Actual Execution Plan option enabled, your query results will return with an additional tab called **Execution plan**. Clicking the tab will return the visual representation of the execution plan that was used while executing your query. Figure 2-27 shows an example of an actual execution plan.

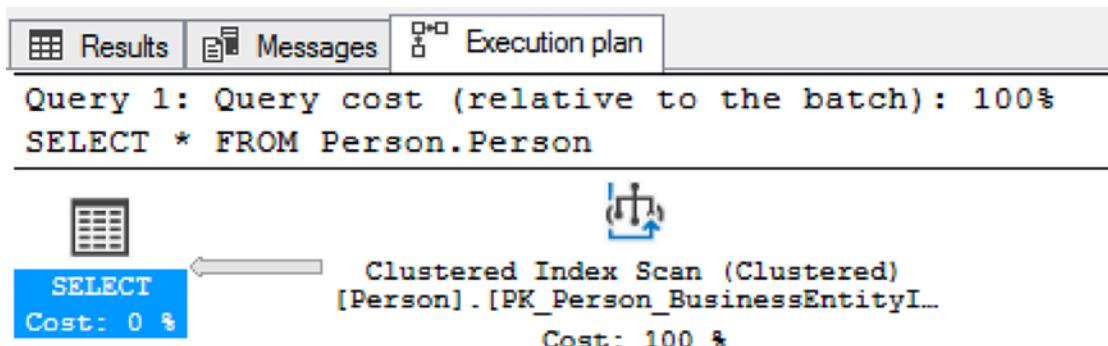


Figure 2-27. Execution plan

With the actual execution plan opened, we can access the per-query wait statistics by right-clicking the first operator, which in the case of Figure 2-28 is the **SELECT** operator, and selecting **Properties**. This will open up the execution plan properties window inside SQL Server Management Studio and reveals a wealth of information, like the degree of parallelism used or the number of rows processed, about the query execution and the various properties of the operator we selected.

Properties	
SELECT	
Misc	
Cached plan size	24 KB
CardinalityEstimationModelVersion	120
CompileCPU	0
CompileMemory	152
CompileTime	0
Degree of Parallelism	1
Estimated Number of Rows	19972
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	2.87192
MemoryGrantInfo	
Optimization Level	TRIVIAL
OptimizerHardwareDependentProperties	
QueryHash	0x88000AA488630C29
QueryPlanHash	0xF27FB7B362CFE7E3
QueryTimeStats	
RetrievedFromCache	true
SecurityPolicyApplied	False
Set Options	ANSI_NULLS: True, ANSI_PADDING: 1
Statement	SELECT * FROM Person.Person
WaitStats	

Figure 2-28. Execution plan properties

Since we are interested in wait statistics, the most interesting part of the execution plan properties is recorded all the way at the bottom of the properties window. When expanding the **WaitStats** properties you are able to see all of the wait types, and wait times, this specific query ran into while executing. Figure 2-29 shows the per-query wait statistics for this specific example query.

WaitStats	
WaitCount	3
WaitTimeMs	2
WaitType	PAGEIOLATCH_SH
WaitCount	3819
WaitTimeMs	2
WaitType	MEMORY_ALLOCATION_EXT
WaitCount	5825
WaitTimeMs	913
WaitType	ASYNC_NETWORK_IO

Figure 2-29. Per-query wait statistics in the execution plan properties

In this case our query encountered three different wait types while executing: PAGEIOLATCH_SH, MEMORY_ALLOCATION_EXT and ASYNC_NETWORK_IO. For each of these wait types you can see how much time was spent waiting, and how many times we waited on the wait type. This information can be very useful when looking at what an individual query encounters in terms of wait statistics during its execution, and perhaps, can give you some insights in how you can tune the performance of the query. For instance, if you see that a query frequently runs into storage-related wait statistics, it might be worth it to investigate how you can minimize storage access for that specific query so it can execute faster.

One thing that I like to point out again, the per-query wait statistics are only recorded in the actual execution plan! The reason why I mention this again is that the actual execution plan is only available by enabling it before execution of a query. There is no other way to access an actual execution plan, not even through the Query Store as we will see in the next chapter. As a matter of fact, the execution plans that are stored in the plan cache of SQL Server are the estimated execution plans and not the actual plans. This means that if you are expecting to retrieve per-query wait statistics through the plan cache you are going to be disappointed.

Thankfully, even though the Query Store feature does not record the actual execution plan, it does (with the release of SQL Server 2017) record wait statistics, and other query runtime information, together with the estimated execution plan. Mixing that information together means that through the Query Store we can look back in time and see what queries encountered in terms of wait statistics!

Summary

In this chapter we reviewed the various ways we can access information about wait statistics. We took an in-depth look at some of the most important DMVs regarding wait statistics: `sys.dm_os_wait_stats`, `sys.dm_os_waiting_tasks`, `sys.dm_exec_requests`, and `sys.dm_exec_session_wait_stats`. I described their functions and the data they returned, and gave you some example queries you can use against those DMVs. We also went through an example scenario where we combined some of the DMVs to analyze what was slowing down the SQL Server in the example. The steps shown in the example are a good way to analyze performance problems on your system when they are occurring. Briefly, we looked at the Windows Performance Monitor, or Perfmon, and how you can access wait statistics information from inside it. After that we took a good look at Extended Events and how you can use them to capture wait-related information for specific queries or sessions using the Extended Events GUI or T-SQL. We ended the chapter by looking at execution plan recorded wait statistics.

CHAPTER 3

The Query Store

With the release of SQL Server 2016, Microsoft introduced an entirely new method to analyze and troubleshoot query performance: the Query Store. The Query Store is often marketed as the “flightrecorder” of SQL Server in that it gives insights into when queries are being executed, how well they performed, and what execution plan was used during execution of the query. While the Query Store did not initially expose wait statistics that were encountered during query execution, the release of SQL Server 2017 included that much awaited addition.

Since the Query Store is such a gamechanger in terms of query performance analysis and tuning, I believe it deserves some additional attention so you can get the most from this feature.

What Is the Query Store?

The Query Store feature was first released in SQL Server 2016 and had a goal of exposing query performance in an easier and more accessible way. Before the Query Store, query performance analysis was a challenging and time-consuming process that requires very thorough knowledge of how SQL Server processes queries and how you can analyze information through the various DMVs. While experience and knowledge of query execution is still very helpful, the Query Store helps you access the performance information you need in a more accessible and visual manner.

The Query Store is integrated directly inside the SQL Server engine. This means that it can capture and analyze query executing where it is occurring. This is a major difference compared to query analysis through other methods, like the execution plan cache, where query runtime information is only available at a later stage. Another advantage of the Query Store is that it persists query runtime information to disk. This means you can build up a history of runtime metrics for your queries and allows easier comparison between historic runtime statistics and current ones. This is a difference

compared to the DMVs, which only records information for the time SQL Server is running and flushes all recorded information at a restart of SQL Server, meaning you start back with all the counters at 0 again. To continue the comparison of the Query Store vs. DMVs, while all the DMVs record information on the entire SQL Server Instance level, the Query Store allows you to capture query runtime metrics on a per-databases basis. This makes analyzing performance for a specific database inside an Instance with multiple databases considerably easier and quicker.

There are more advantages of using the Query Store, for instance, the easy forcing of execution plans, but since this is a book about wait statistics, I will not cover everything that the Query Store has to offer. If you are interested in more in-depth look of the Query Store, I wrote a series of articles that describes just about everything the Query Store is capable of and it's available here: www.red-gate.com/simple-talk/sql/database-administration/the-sql-server-2016-query-store-overview-and-architecture/.

Query Store Architecture

To give you a good idea on how the Query Store works, I created the image in Figure 3-1 that shows how query runtime information is recorded and stored inside the Query Store. This knowledge is important if you want to add the Query Store as a tool to analyze query performance and wait statistics.

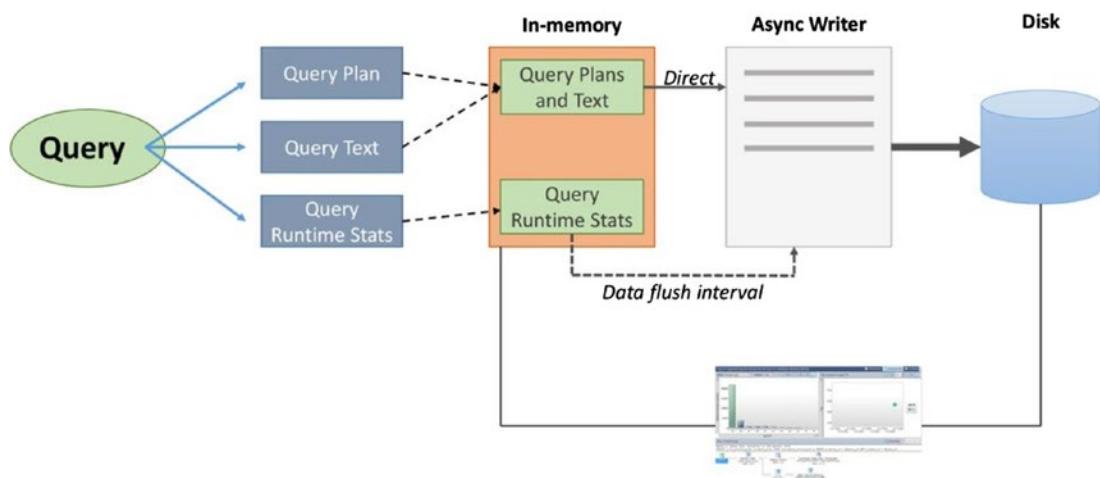


Figure 3-1. Query Store architecture

As soon as a query is starting its execution, and you have the Query Store enabled on the database the query is being executed against, the Query Store splits the query runtime information into three parts: the query execution plan, the query text, and the runtime statistics of the query execution. Both the execution plan and the query text are immediately recorded in the Query Store during the compilation of the execution plan. The runtime statistics (including wait statistics information) are only available after execution of the query, meaning they will be added to the Query Store after query execution.

All of the Query Store metrics are first stored inside a reserved memory area of SQL Server. This means they are directly available through queries or the built-in reports, but are not hardened to disk yet (exception is new execution plans, these are hardened directly). Based on a configurable setting inside the Query Store called the “Data flush interval” you can configure how fast the Query Store should harden the information to disk.

We can access all the information the Query Store records through two methods: Query Store DMVs and the built-in reports. To expose the wait statistics inside the Query Store, we will be using both options, though I believe the DMV approach is more useful when looking at the wait statistics data.

How Wait Statistics Are Processed in the Query Store

Before we can get started looking at how we can access the wait statistics information recorded in the Query Store, we need to look at how the Query Store processes wait statistics, since it is different compared to the process I described in Chapter 1, “Wait Statistics Internals,” of this book. In that chapter I described that the SQL Server engine keeps track of how long a query spends waiting on a specific resource, or wait type. This information is recorded at a very granular level; for instance, the PAGEIOLATCH_SH wait type indicates that the query is waiting for data pages to be read from disk to the buffer cache and a PAGEIOLATCH_EX that the query is waiting for a data page to be moved to the buffer cache for modification. Instead of recording wait times on such a detailed level, the team behind the development of the Query Store decided that a more high-level wait type overview was required to avoid performance and resource utilization overhead. Ultimately, they chose to group various wait types together into categories and record wait times on a category level instead of individual wait types.

This means that searching for specific wait types inside the Query Store recorded wait statistics is not possible and wait types that share the same category cannot be distinguished from each other. To give you an example, both the CMEMTHREAD and the RESOURCE_SEMAPHORE wait types (of which you will learn more in Chapter 6, “IO-Related Wait Types”) are recorded inside the Memory category inside the Query Store, even though both wait types indicate different things.

Table 3-1 shows the mapping between wait types and wait categories that are used in the Query Store. The table is by no means a complete overview of the mappings but should give you a good idea where to expect a certain wait type.

Table 3-1. Mapping Between wait types and Categories

Wait Category	Associated Wait Types
Unknown	Unknown
CPU	SOS_SCHEDULER_YIELD
Worker thread	THREADPOOL
Lock	LCK_M_%
Latch	LATCH_%
Buffer latch	PAGELATCH_%
Buffer IO	PAGEIOLATCH_%
Compilation*	RESOURCE_SEMAPHORE_QUERY_COMPILE
SQL CLR	CLR%, SQLCLR%
Mirroring	DBMIRROR%
Transaction	XACT%, DTC%, TRAN_MARKLATCH_%, MSQL_XACT_%, TRANSACTION_MUTEX
Idle	SLEEP_%, LAZYWRITER_SLEEP, SQLTRACE_BUFFER_FLUSH, SQLTRACE_INCREMENTAL_FLUSH_SLEEP, SQLTRACE_WAIT_ENTRIES, FT_IFTS_SCHEDULER_IDLE_WAIT, XE_DISPATCHER_WAIT, REQUEST_FOR_DEADLOCK_SEARCH, LOGMGR_QUEUE, ONDemand_TASK_QUEUE, CHECKPOINT_QUEUE, XE_TIMER_EVENT
Preemptive	PREEMPTIVE_%

(continued)

Table 3-1. (continued)

Wait Category	Associated Wait Types
Service broker	BROKER_% (but not BROKER_RECEIVE_WAITFOR)
Tran Log IO	LOGMGR, LOGBUFFER, LOGMGR_RESERVE_APPEND, LOGMGR_FLUSH, LOGMGR_PMM_LOG, CHKPT, WRITELOGF
Network IO	ASYNC_NETWORK_IO, NET_WAITFOR_PACKET, PROXY_NETWORK_IO, EXTERNAL_SCRIPT_NETWORK_IOF
Parallelism	CXPACKET, EXCHANGE
Memory	RESOURCE_SEMAPHORE, CMEMTHREAD, CMEMPARTITIONED, EE_PMOLOCK, MEMORY_ALLOCATION_EXT, RESERVED_MEMORY_ALLOCATION_EXT, MEMORY_GRANT_UPDATE
User wait	WAITFOR, WAIT_FOR_RESULTS, BROKER_RECEIVE_WAITFOR
Tracing	TRACEWRITE, SQLTRACE_LOCK, SQLTRACE_FILE_BUFFER, SQLTRACE_FILE_WRITE_IO_COMPLETION, SQLTRACE_FILE_READ_IO_COMPLETION, SQLTRACE_PENDING_BUFFER_WRITERS, SQLTRACE_SHUTDOWN, QUERY_TRACEOUT, TRACE_EVTNOTIFF
Full text search	FT_RESTART_CRAWL, FULLTEXT_GATHERER, MSSEARCH, FT_METADATA_MUTEX, FT_IFTSHC_MUTEX, FT_IFTSIM_MUTEX, FT_IFTS_RWLOCK, FT_COMPROWSET_RWLOCK, FT_MASTER_MERGE, FT_PROPERTYLIST_CACHE, FT_MASTER_MERGE_COORDINATOR, PWAIT_RESOURCE_SEMAPHORE_FT_PARALLEL_QUERY_SYNC
Other disk IO	ASYNC_IO_COMPLETION, IO_COMPLETION, BACKUPIO, WRITE_COMPLETION, IO_QUEUE_LIMIT, IO_RETRY
Replication	SE_REPL_% , REPL_% , HADR_% (but not HADR_THROTTLE_LOG_RATE_GOVERNOR), PWAIT_HADR_% , REPLICA_WRITES, FCB_REPLICA_WRITE, FCB_REPLICA_READ, PWAIT_HADRSIM
Log rate governor	LOG_RATE_GOVERNOR, POOL_LOG_RATE_GOVERNOR, HADR_THROTTLE_LOG_RATE_GOVERNOR, INSTANCE_LOG_RATE_GOVERNOR

Accessing Wait Statistics Through the Query Store Reports

The most user-friendly way to view the wait statistics that are available inside the Query Store is through the built-in reports that are available inside SQL Server Management Studio after you enabled the Query Store feature on a database. Figure 3-2 shows the default, built-in, Query Store reports that are available at the time of writing this book.



Figure 3-2. *Query Store reports*

As you can see in Figure 3-2, there are no dedicated wait statistics reports (yet). Instead of a dedicated report, we can view the wait categories as a query encountered by specifically selecting the **Wait Time (ms)** metric inside the following three reports:

- Regressed Queries
- Top Resource Consuming Queries
- Queries with High Variation

To view the wait categories in any of the preceding reports, you first need to configure the metric to Wait Time (ms), as shown in Figure 3-3.

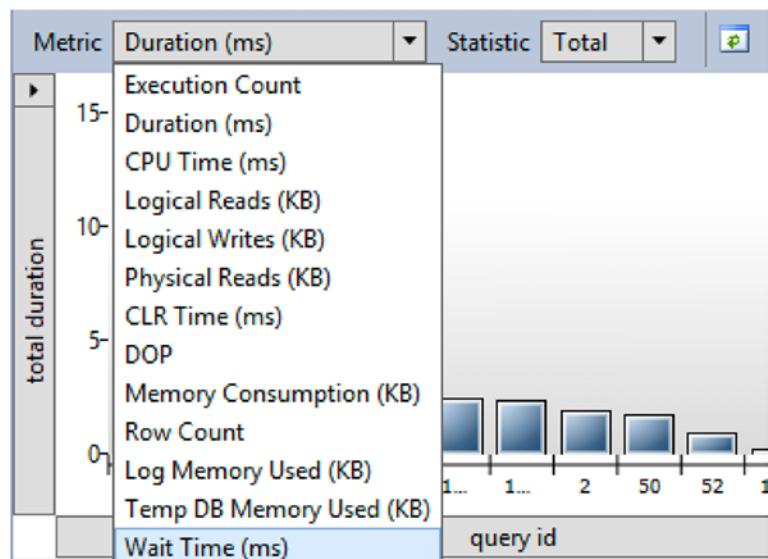


Figure 3-3. Configuring the metric to Wait Time (ms)

This changes the graph to return (by default) the top 25 queries order by the total wait time.

After changing the metric, you can mouseover on any of the queries that are shown in the graph to retrieve the wait category information, as shown in Figure 3-4.

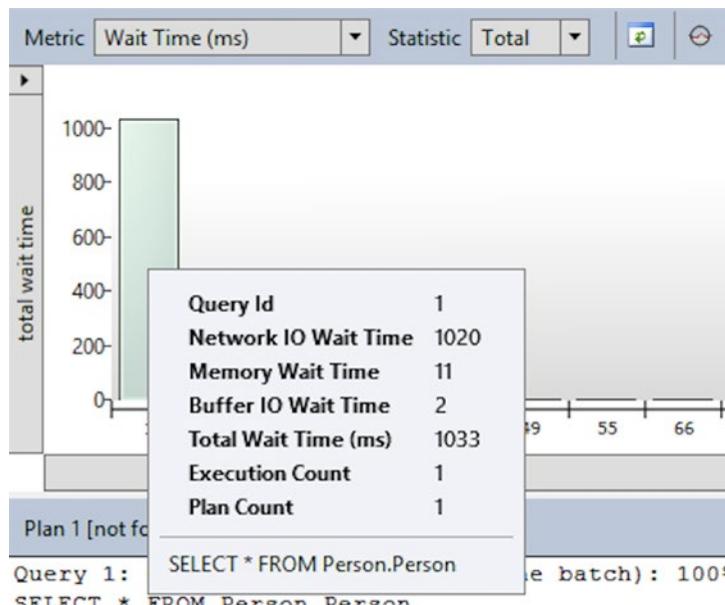


Figure 3-4. Wait categories exposed in the Query Store

As you can see from Figure 3-4, this specific query ran into various wait categories: Network IO, Memory, and Buffer IO. We can see the wait time per category and the total wait time across all of the categories, but as mentioned earlier, we have no idea about which exact wait types were encountered by the query.

Accessing Wait Statistics Through the Query Store DMVs

While the built-in Query Store reports are definitely helpful in visibly identifying queries with high wait times, we can easily generate far more information by using a new Query Store DMV that is available in SQL Server 2017: **sys.query_store_wait_stats**. The columns that are returned by querying the DMV mainly show various statistics related to the wait types of the various wait categories for a specific execution plan ID.

The Query Store uses its own unique identifiers for queries, execution plans, and runtime intervals. With that in mind, you can identify queries by looking up the Query ID inside the Query Store, or execution plans by searching on the Plan ID.

Figure 3-5 shows the different statistics that are recorded for each execution plan ID, split up into different wait categories.

wait_category_desc	execution_type	execution_type_desc	total_query_wait_time_ms	avg_query_wait_time_ms	last_query_wait_time_ms	min_query_wait_time_ms	max_query_wait_time_ms
Latch	0	Regular	21	21	21	21	21
Buffer IO	0	Regular	5	5	5	5	5
Idle	0	Regular	9	9	9	9	9
Network IO	0	Regular	54	54	54	54	54
Parallelism	0	Regular	293	293	293	293	293
Memory	0	Regular	3	3	3	3	3
Unknown	0	Regular	4	1.33333333333333	2	0	2
CPU	0	Regular	18	6	0	0	17
Idle	0	Regular	15	5	5	3	6
Network IO	0	Regular	203	67.66666666666667	35	35	116
Parallelism	0	Regular	854	284.6666666666667	245	217	352

Figure 3-5. Wait categories statistics inside sys.query_store_wait_stats

While we can just query the sys.query_store_wait_stats DMV and look at the various statistics for each, or a specific, execution plan ID, we can get far more information by joining the various Query Store DMVs together.

As an example, the following query joins various Query Store DMVs to return an overview of queries that encounter high total wait times.

```

SELECT
    qsqs.plan_id,
    qsq.query_id,
    qsqs.runtime_stats_interval_id,
    qsqt.query_sql_text,
    qsqs.wait_category_desc,
    qsqs.total_query_wait_time_ms
FROM sys.query_store_wait_stats qsqs
INNER JOIN sys.query_store_plan qsp
ON qsqs.plan_id = qsp.plan_id
INNER JOIN sys.query_store_query qsq
ON qsp.query_id = qsq.query_id
INNER JOIN sys.query_store_query_text qsqt
ON qsq.query_text_id = qsqt.query_text_id
ORDER BY qsqs.total_query_wait_time_ms DESC

```

The results of this query can be seen in Figure 3-6.

	plan_id	query_id	runtime_stats_interval_id	query_sql_text	wait_category_desc	total_query_wait_time_ms
1	8	8	2	select top 1000 * from sales.SalesOrderDetail order ...	Parallelism	854
2	10	10	2	select top 1001 * from sales.SalesOrderDetail order ...	Parallelism	655
3	8	8	1	select top 1000 * from sales.SalesOrderDetail order ...	Parallelism	293
4	10	10	2	select top 1001 * from sales.SalesOrderDetail order ...	Parallelism	284
5	14	14	2	SELECT * FROM sys.query_store_wait_stats qsqs...	Network IO	206
6	8	8	2	select top 1000 * from sales.SalesOrderDetail order ...	Network IO	203
7	10	10	2	select top 1001 * from sales.SalesOrderDetail order ...	Network IO	159
8	15	15	2	SELECT qsqs.plan_id, qsq.query_id, qsqt.que...	Network IO	120
9	17	17	2	SELECT qsqs.plan_id, qsq.query_id, qsqs.ru...	Network IO	105
10	13	13	2	select * from sys.query_store_wait_stats	Network IO	68
11	8	8	1	select top 1000 * from sales.SalesOrderDetail order ...	Network IO	54
12	10	10	2	select top 1001 * from sales.SalesOrderDetail order ...	Network IO	45
13	16	16	2	SELECT qsqs.plan_id, qsq.query_id, qsqt.que...	Network IO	41

Figure 3-6. Wait statistics information from the Query Store

Using the preceding query, we can immediately see that the SELECT query against the Sales.SalesOrderDetail table spends most of its waiting time on parallelism-related wait types.

CHAPTER 3 THE QUERY STORE

Modifying the query a little bit by filtering on the queries Query ID allows us to zoom in on a specific query and analyze its wait behavior. Following is the modified query, and you'll see that I also added some additional columns that return various useful statistics.

```
SELECT
    qsqs.plan_id,
    qsq.query_id,
    qsqs.runtime_stats_interval_id,
    qsqt.query_sql_text,
    qsqs.wait_category_desc,
    qsqs.total_query_wait_time_ms,
    qsqs.avg_query_wait_time_ms,
    qsqs.last_query_wait_time_ms
FROM sys.query_store_wait_stats qsqs
INNER JOIN sys.query_store_plan qsp
ON qsqs.plan_id = qsp.plan_id
INNER JOIN sys.query_store_query qsq
ON qsp.query_id = qsq.query_id
INNER JOIN sys.query_store_query_text qsqt
ON qsq.query_text_id = qsqt.query_text_id
WHERE qsq.query_id = 8
ORDER BY runtime_stats_interval_id ASC
```

Figure 3-7 shows the output from running this version of the query.

plan_id	query_id	runtime_stats_interval_id	query_sql_text	wait_category_desc	total_query_wait_time_ms	avg_query_wait_time_ms	last_query_wait_time_ms
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Unknown	4	4	4
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Latch	21	21	21
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Buffer IO	5	5	5
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Idle	9	9	9
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Network IO	54	54	54
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Parallelism	293	293	293
8	8	1	select top 1000 * from sales.SalesOrderDetail or...	Memory	3	3	3
8	8	2	select top 1000 * from sales.SalesOrderDetail or...	Unknown	4	1.33333333333333	2
8	8	2	select top 1000 * from sales.SalesOrderDetail or...	CPU	18	6	0
8	8	2	select top 1000 * from sales.SalesOrderDetail or...	Idle	15	5	5

Figure 3-7. Wait statistics information for a specific query

In the query and the query results shown in Figure 3-7, you can see I added an additional column called **runtime_stats_interval_id**. The wait the Query Store groups runtime, and also wait time, metrics is by aggregating them based on intervals. By default,

these intervals are one-hour blocks, meaning that the wait statistics that are returned for our specific query in the preceding example are the aggregated results of one, or multiple, query executions inside the interval. While you can lower the intervals to smaller time segments by setting the **Statistics Collection Interval** setting inside the Query Store properties, this can have a negative impact on the performance of your SQL Server Instance, so be careful when changing this setting.

I have only shown you a few examples of how you can retrieve wait statistics-related metrics from inside the Query Store. With all the information that the Query Store collects there is a whole ocean full of other information to combine with the wait statistics metrics. For instance, you can filter on specific wait categories, or detect if queries generate different execution plans and what their impact is on the waits for that specific query. I definitely recommend everyone that is using SQL Server 2016 or higher to enable the Query Store and explore all the amazing metrics it collects.

While the Query Store is only available from SQL Server 2016 and higher, William Durkin (@sql_williamd on Twitter) and myself released a project called Open Query Store which emulates Query Store data collection for SQL Server versions lower than 2016. The project is completely open source and free and available through the project's GitHub page at <https://github.com/OpenQueryStore/OpenQueryStore>.

Summary

In this chapter we looked at a new query performance and analysis feature that was introduced in SQL Server 2016: the Query Store. We looked at how the Query Store works underneath the covers and how we can access query wait statistics information in the built-in reports of the SQL Server 2017 version of the Query Store.

Finally, we looked at accessing Query Store wait statistics using the new Query Store DMVs and showed some example queries that can help you get started on querying the Query Store.

CHAPTER 4

Building a Solid Baseline

In Chapter 2, “Querying SQL Server Wait Statistics,” we spent a great deal of time describing and using the various methods of accessing wait statistics information. Most of those methods focused on using that information for detecting performance problems that are presently occurring. While it is possible to find the exact cause of the performance problem using these real-time methods, it requires a deep knowledge of the various wait types and—most important—experience in the performance of your SQL Server. If you are managing only one SQL Server instance, you can get yourself familiar with the way it reacts under different circumstances relatively quickly. If you are managing hundreds of SQL Server instances, getting yourself familiar with the way they perform is impossible. Because SQL Server wait statistics are largely based on the workload of your SQL Server instance, no two SQL Server instances will have the same wait times for the same wait types. This makes detecting possible problems difficult because we can’t just say “because the CXPACKET wait type has a wait time of 20,000 milliseconds we are having a problem.” It all depends on the configuration and workload of your system. One SQL Server instance can have 20,000 milliseconds (20 seconds) of wait time every minute spent on the CXPACKET wait type and experience no performance problems, while another instance has 1,000 milliseconds of wait time and users are constantly complaining about performance.

If we want to perform an in-depth analysis of wait statistics, or any performance-related data, we need a method of collecting performance-related metrics and giving them meaning. Just detecting that you spend 10,000 milliseconds waiting for resources doesn’t mean anything, since we do not know if it caused performance problems or not. Yes, maybe your users are complaining that performance is horrible while you notice that 10,000 milliseconds wait, but there is no way to be sure if that wait is actually causing the performance problems your users are experiencing. In those cases we frequently take a guess and just assume that the specific resource wait is the source of the performance problem. I learned, after talking to many DBAs across the world, that

DBAs do not like to take guesses at what's slowing down our SQL Server instances. We want to be certain that the source of the performance problem actually is that resource wait. This is where baselines can help.

Baselines will help you give meaning to performance metrics by providing you with a definition of what the normal situation is on your system. Without a baseline of our system, we have no idea if it is running optimally or terribly slow. Baselines are incredibly important; in fact, they are so important that I decided to write a complete chapter about them in this book. Without a solid baseline, your measurements mean nothing! Even though this book focuses on wait statistics, baselines can be used for every performance-related metric you can capture on your system, giving you a valuable method of performance analysis.

If you read through the previous chapter concerning the Query Store, you might be tempted to think that the Query Store can handle all your baseline needs (if you are running SQL Server 2016 or higher). While the Query Store absolutely provides a very useful tool in capturing and monitoring the performance of your query, it doesn't necessarily provide a performance overview of your entire SQL Server Instance. I personally consider the Query Store an addition to the regular process of capturing and monitoring baselines, not a replacement.

What Are Baselines?

If we look up the word "baseline" in the Oxford dictionary, we would get the following definition: "A minimum or starting point used for comparisons." This sentence captures the essence of a baseline perfectly by using the important words "starting point" and "comparisons." A baseline will generally be the starting point, or in our case, measurement, that we will compare later measurements against. Ideally, we will capture our baseline measurement in a normal or standard situation. If we perform the same measurement again at a later point in time, we can compare that measurement against the baseline. If our measurements are not the same during the comparison, something might have changed.

Even if you do not yet use baseline comparisons when analyzing performance, you are still working with baselines constantly whether you realize it or not. For instance, if you receive a salary every first day of the month, that would be your normal situation or, in the context of this chapter, your baseline. If for some reason you didn't receive your salary on the first day of the month, you would notice a deviation compared to the

baseline. This might be a reason to investigate why you didn't receive your salary on time. Maybe the day changed from the first to the fifth of the month or, in the worst case, the company you work for can't pay your salary anymore. In any of these cases, there are actions we can take: either accept the change, and by doing so create a new baseline, or revert the situation back to the baseline state again. Figure 4-1 shows this process.

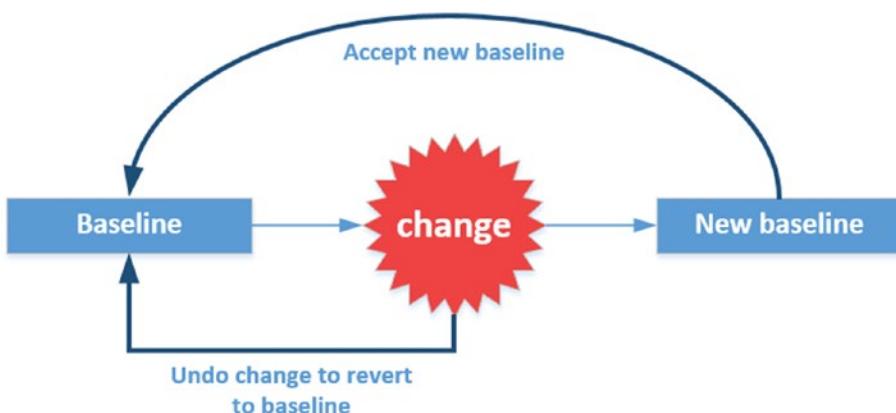


Figure 4-1. Changes impact baseline

Defining and maintaining baselines is an iterative process. Every change occurring on your system can impact your baseline. A new release of the application you are using might change a number of queries, or your company might grant a new department access to the database, increasing the number of connections. In both these examples we would need to make adjustments to the baseline, since the normal situation would have changed.

All of this adjusting and measuring baselines with every change to the system sounds like a lot of work, and sometimes it is. But believe me—the benefits of having a baseline far outweigh the costs. Baselines will help you detect problems far faster than just looking at a single measurement, and in the case of wait statistics, it is the only way to find a reliable, definitive answer to your problem. Let's use a more technical example to illustrate this using DBA Jim.

Jim maintains a SQL Server instance that hosts a single-user database. This database is used by every sales person in the company and records every financial transaction between the company and its customers. Users can access the database through Application X. Application X is currently running version 2.4 and is very stable. Performance is good, users are happy, and the money keeps rolling in. Sounds great,

right? One day a consultant walks in and wants to upgrade Application X to the brand new 3.0 version. The update to version 3.0 was a breeze and completed without any problems, and all the users love the new features.

Two days later the phone is ringing, Chris's manager just received word from the sales team that the performance in version 3.0 is horrible and he demands it get resolved right away.

Thankfully, Jim knew the importance of a baseline, and he created one before the upgrade to version 3.0. Using the version 2.4 baseline, Jim compares the measurements in the baseline to the measurements done in version 3.0 and immediately spots a large difference in the lock wait time measurements. Since other measurements remain more or less the same compared to the 2.4 baseline, Jim focuses on long-running locks and identifies an update query that is locking a table. He rolls back the query, and the situation returns to normal. He then contacts the application's vendor and learns this behavior was a result of a bug in the software.

Now this example might sound a little far-fetched, but it is actually a simplified version of the method I use almost every day when measuring the impact of changes or analyzing performance problems. If Jim didn't have a baseline of the lock wait time and just queried the lock wait time after the change to version 3.0, he wouldn't know that the wait time had increased, since he had nothing to compare against. He might have chosen to look at other metrics instead of the lock wait time and would have wasted valuable time and money.

The message here is simple: baselines will help you detect abnormal situations and resolve performance problems faster!

Visualizing Your Baselines

Baselines are frequently visualized through graphs. The big advantage of turning your baseline measurements into graphs is that a graph can make it easier for you to detect those measurements that have the highest increase or decrease compared to your baseline. Also, visualizing your data might help you get your point across easier if you have to prove that a specific configuration is impacting performance. For example, you need to convince your storage administrator that the change in the storage configuration has impacted your performance. If you can hand him a graph that shows the normal behavior compared to the behavior after the change, he might be more inclined to help. The graph in Figure 4-2 shows the baseline measurements compared to the measurements done at a later time.

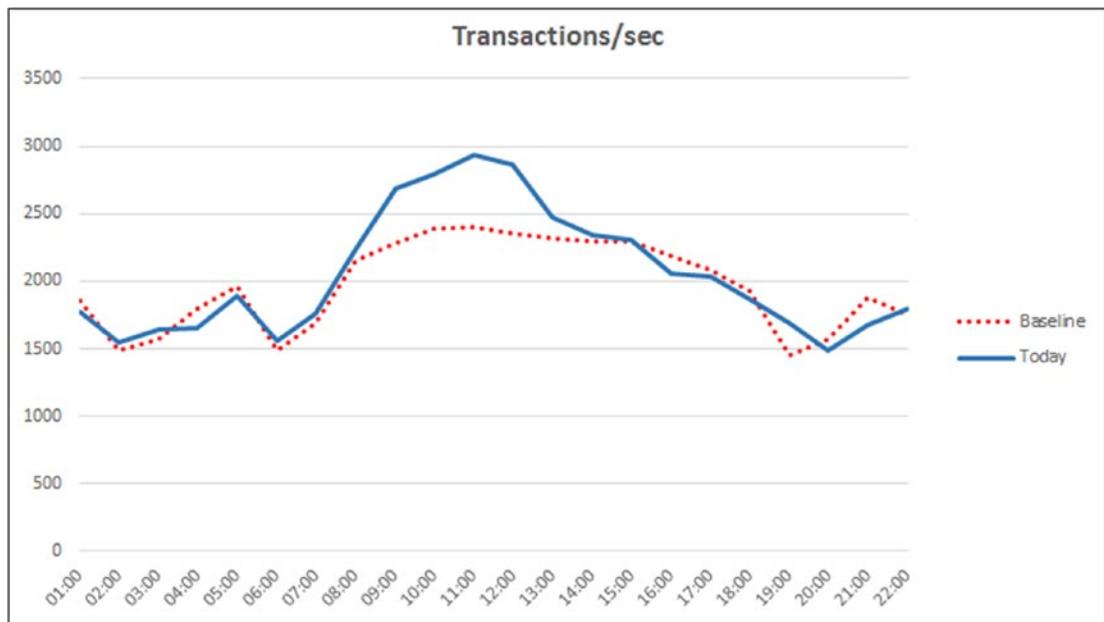


Figure 4-2. Example of a baseline graph

As you can see in Figure 4-2, you would be able to identify the potential problem very quickly. Apparently, between 08:00 and 12:00 the number of transactions per second is higher than the normal situation, and it might be worth taking the time to investigate.

Baseline Types and Statistics

We will frequently use different types of baselines depending on the information we are interested in. There usually isn't one single baseline to fit all our needs, especially not when you are using baselines for performance-troubleshooting purposes. For instance, we can create a baseline for every single wait type, or we can choose to create only baselines for wait types that impact our system the most. We can also choose to create a baseline for specific days or time segments, like business hours, and create another for after business hours.

Next to selecting or limiting the measurements we want to have baselines for, we also have to make choices on how we calculate our baselines. These choices involve some math and usually require calculating averages. In many cases our baseline consists of an average of many data points, depending on how many measurements you performed. If you collect measurements for a long period of time and calculate an average value from

CHAPTER 4 BUILDING A SOLID BASELINE

those measurements, you can create a more reliable baseline than you can when you only have one day's worth of measurements. Creating a baseline based on averages also has its disadvantages. The most important one is that averages are heavily influenced by skewed data. Without going too deep into statistical details, skewed data means that there are very high or very low values that impact your average. Say, for instance, that a group of students took an exam and we wanted to see how the group performed by calculating the average result of the exam (the students are rated between 1 and 10, 1 being very poor and 10 being excellent; a 6 or higher is required to pass the exam).

Figure 4-3 shows the exam results in a graph.

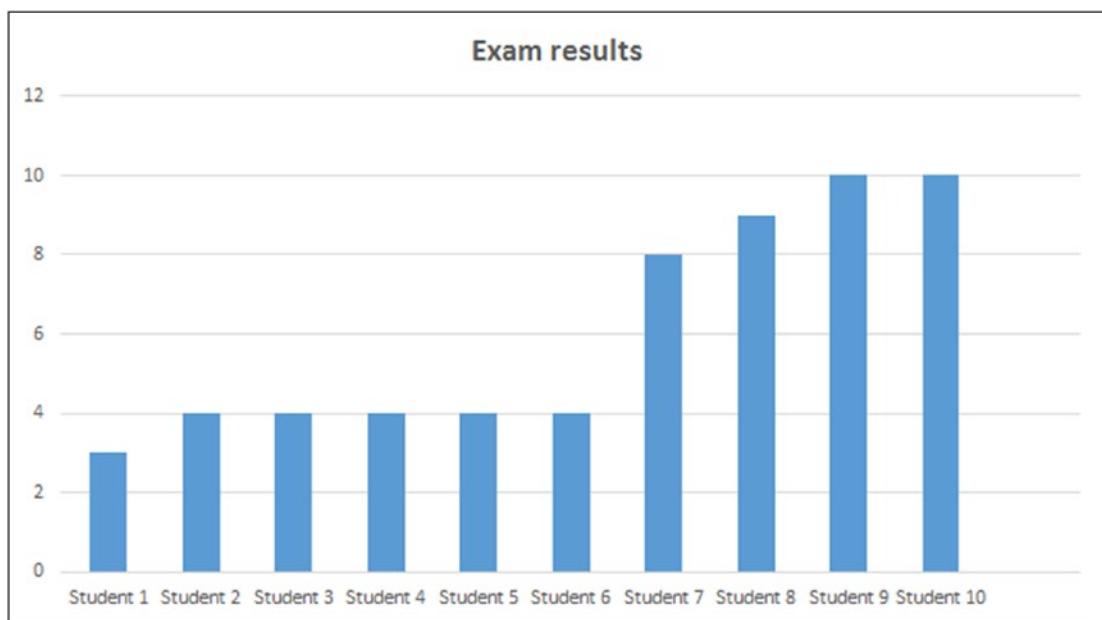


Figure 4-3. Exam results

As you can see in Figure 4-3, only four students scored higher than the required 6 to pass the exam. The rest of the group scored way below the requirement. However, if we look at the average performance of the group, they actually didn't do too badly by scoring a 6. We could conclude the group is performing well enough by getting an average score of 6, but then we would miss that the majority performed badly and only four students actually passed the exam. This information is important to keep in mind when you are dealing with average baselines. If you see a spike in your average baseline, it is always something you should investigate, because it impacts your baseline.

There are statistical methods available to deal with skewed data and averages, one being the trimmed (or truncated) mean. This method removes x percent of the highest and lowest measurements in your series, creating a more stable average. We won't go any deeper into the trimmed mean for baseline use, but if you want to learn more about it, I suggest you read Bob Newstadt's blog post at www.sqlteam.com/article/computing-the-trimmed-mean-in-sql. Even though the article is pretty dated, it shows a method of calculating the trimmed mean using T-SQL.

Baseline Pitfalls

Hopefully the previous section convinced you that baselines are important, but before you go and capture every performance metric and convert it into a baseline measurement, there are some pitfalls you will want to avoid.

Too Much Information

Even though you are free to baseline everything in your system, this is generally considered a bad idea. Gathering too much information can blind you in your search for answers. If you have to compare 100,000 different metrics against your baseline every time a performance problem occurs, you are wasting time. The advice here is to keep your baselines small, including only performance metrics that matter the most for your system. For instance, you can include performance metrics related to Availability Groups, but if your system doesn't use this feature, then there is no use including them.

Know Your Metrics

Another important aspect in the selection of performance metrics is understanding. If you do not understand what a performance metric represents, it can be very difficult to formulate a correct conclusion, or it can even lead you in the wrong direction.

Focus on the Big Measurement Changes

When comparing measurements against a baseline, always focus on the big increases or decreases. Especially for wait statistics, very small increases in wait time (1-2%) aren't a cause for concern. If one of your wait time measurements goes up 20%, that would be a good signal to start investigating.

Use Fixed Intervals

When capturing wait statistics information you should always use a fixed interval. If we were to capture wait times at random, it would be almost impossible to build a reliable baseline. It would be like comparing apples against oranges. The best way to automate the capture of wait statistics information is by using the SQL Server Agent and setting it to a fixed interval, like every 15 minutes.

Building a Baseline for Wait Statistics Analysis

Now that we have familiarized ourselves with baselines, let's get to work and create a baseline we can use in our wait statistics analysis. As I mentioned at the beginning of this chapter, baselines are incredibly important if you want to analyze performance problems using wait statistics. Nobody has the same wait types and wait times compared to your system, so it's up to you to create a baseline you can compare against.

In this section I will show you a method I use to create, maintain, and compare baselines and measurements. This does not necessarily mean this is the right way to do it, and you might find other methods better suited to your needs.

Since we are going to capture SQL Server wait statistics measurements, I prefer to store my measurements inside a separate database named "Baseline." This way my measurement information doesn't get stored somewhere between user tables. Since wait statistics are logged at the SQL Server instance level, it makes sense to create a separate measurement database inside every SQL Server instance. Figure 4-4 shows you my baseline database inside the SQL Server Management Studio.

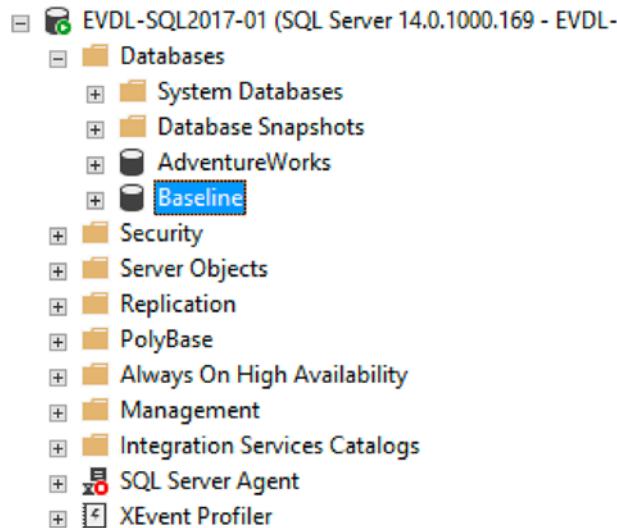


Figure 4-4. Baseline database

You can create the database yourself using the script in Listing 4-1, making sure to change the file locations. The database data file will be 1.5 GB when it gets created, which gives enough free space to capture weeks of wait statistics information.

Listing 4-1. Creating the Baseline database

```
-- Create Baseline database
CREATE DATABASE [Baseline]
ON PRIMARY
(
    NAME = N'Baseline', FILENAME = N'E:\Data\baseline_data.mdf' ,
    SIZE = 1536000KB , FILEGROWTH = 10%
)
LOG ON
(
    NAME = N'Baseline_log', FILENAME = N'E:\Log\baseline_log.ldf' ,
    SIZE = 102400KB , FILEGROWTH = 10%
)
GO

ALTER DATABASE [Baseline] SET RECOVERY SIMPLE
GO
```

We will be using the `sys.dm_os_wait_stats` DMV as the source of our measurements, which means that the table that will hold our measurements must be able to handle the information returned from the DMV. We will not only store the wait types and wait times but will also add additional information to enrich the data so that we can easily create multiple baselines.

Listing 4-2 shows the query you can use to create a table, named `WaitStats`, to hold the wait statistics information we will use for creating our baselines.

Listing 4-2. Create a wait statistics table

```
USE [BaseLine]
GO

CREATE TABLE WaitStats
(
    ws_ID INT IDENTITY(1,1) PRIMARY KEY,
    ws_DateTime DATETIME,
    ws_Day INT,
    ws_Month INT,
    ws_Year INT,
    ws_Hour INT,
    ws_Minute INT,
    ws_DayOfWeek VARCHAR(15),
    ws_WaitType VARCHAR(50),
    ws_WaitTime INT,
    ws_WaitingTasks INT,
    ws_SignalWaitTime INT
)
```

As you can read in the listing, we capture the wait type, wait time, signal wait time, and the number of waiting tasks. We also capture the date and time when we log the wait statistics information. We also split the date and time into additional columns to segment the data, making it easier to build specific baselines based on a specific day, hour, month, and so forth, without having to convert the `datetime` data type every time.

Now that we have our table ready, it is time to capture some wait statistics and insert them into our `WaitStats` table. Because the `sys.dm_os_wait_stats` DMV returns cumulative wait times, we have to use a method to only capture the difference in wait

time between two capture moments. If we were to only capture the information directly from the `sys.dm_os_wait_stats` DMV, we would always receive ever-increasing wait times, and that would make comparisons useless. There are two paths we can take to capture the change in wait time between two measurements, and both have their advantages and disadvantages.

The first method, which I call the reset method, will capture the wait statistics information from the `sys.dm_os_wait_stats` DMV and then reset the DMV using the `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` command. The main advantage of this method is that it is very simple to use, as we only need to capture the information, reset it again, and start the same procedure at the next measurement. There is no need to calculate deltas, because after our first measurement the counters are reset to 0. The disadvantage is that the `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` command resets the information inside the `sys.dm_os_wait_stats` DMV. This means that you will lose the cumulative information inside the DMV, information you might not want to lose. Figure 4-5 illustrates this method of wait statistics capturing.



Figure 4-5. Capturing wait statistics using the reset method

The second option, which I named the delta method, involves not using the `DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)` command, but rather calculating the difference, or delta, in wait time between two measurements. The advantage of not using the DBCC command is that you will not lose the cumulative wait times inside the `sys.dm_os_wait_stats` DMV. Its main disadvantage is that it is a lot more complex to calculate the deltas compared to the first method. It usually also involves a `WAITFOR DELAY` command inside the T-SQL script to set the interval. This might mean that if you plan to capture wait statistics information using the SQL Server Agent, you could end up with a SQL Server Agent job that is running almost continuously. Figure 4-6 illustrates the delta option of capture wait statistics.

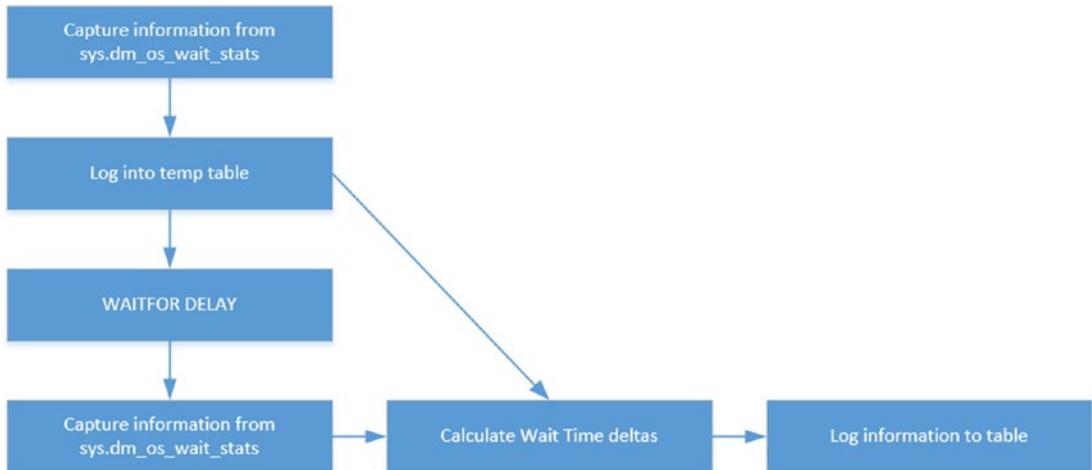


Figure 4-6. Capturing wait statistics using the delta method

There are more methods available with which to capture wait statistics information, but I most frequently see these two, or variations of them. What method you want to use is up to you, as in the end both will return the same results.

Reset Capture Method

The reset wait statistics capture method consists of a single T-SQL script that will capture the information from the `sys.dm_os_wait_stats` DMV followed by a reset of the counters inside the DMV. Listing 4-3 shows the T-SQL script you can use to fill the `WaitStats` table we created earlier.

Listing 4-3. Reset capture method

```

USE [Baseline]
GO

-- Insert Wait Stats into Baseline table
INSERT INTO WaitStats
SELECT
    GETDATE() AS 'DateTime',
    DATEPART(DAY,GETDATE()) AS 'Day',
    DATEPART(MONTH,GETDATE()) AS 'Month',
  
```

```

DATEPART(YEAR,GETDATE()) AS 'Year',
DATEPART(HOUR, GETDATE()) AS 'Hour',
DATEPART(MINUTE, GETDATE()) AS 'Minute',
DATENAME(DW, GETDATE()) AS 'DayOfWeek',
wait_type AS 'WaitType',
wait_time_ms AS 'WaitTime',
waiting_tasks_count AS 'WaitingTasks',
signal_wait_time_ms AS 'SignalWaitTime'
FROM sys.dm_os_wait_stats;

-- Clear sys.dm_os_wait_stats
DBCC SQLPERF ('sys.dm_os_wait_stats',CLEAR)
GO

```

Delta Capture Method

The delta capture method also consists of a single T-SQL script, but it is a little more complex than the reset capture method. It uses a temporary table to store the first measurement, then waits for 15 minutes, performs a second measurement, and calculates the deltas. The result is inserted into the `WaitStats` table. Listing 4-4 shows the T-SQL script you can use if you plan to use this method of collecting wait statistics metrics.

Listing 4-4. Delta capture method

```

USE [Baseline]
GO

-- Check if the temp table already exists
-- if it does drop it.
IF EXISTS

(
SELECT *
FROM tempdb.dbo.sysobjects
WHERE ID = OBJECT_ID(N'tempdb..#ws_Capture')
)
DROP TABLE #ws_Capture;

```

CHAPTER 4 BUILDING A SOLID BASELINE

```
-- Create temp table to hold our first measurement
CREATE TABLE #ws_Capture
(
    wst_WaitType VARCHAR(50),
    wst_WaitTime INT,
    wst_WaitingTasks INT,
    wst_SignalWaitTime INT
);

-- Insert our first measurement into the temp table
INSERT INTO #ws_Capture
SELECT
    wait_type,
    wait_time_ms,
    waiting_tasks_count,
    signal_wait_time_ms
FROM sys.dm_os_wait_stats;

-- Wait for the next measurement
-- In this case we will wait 15 minutes
WAITFOR DELAY '00:15:00'

-- Combine the first measurement with a new
-- measurement and calculate the deltas
-- Write the results into the WaitStats table
INSERT INTO WaitStats
SELECT
    GETDATE() AS 'DateTime',
    DATEPART(DAY,GETDATE()) AS 'Day',
    DATEPART(MONTH,GETDATE()) AS 'Month',
    DATEPART(YEAR,GETDATE()) AS 'Year',
    DATEPART(HOUR, GETDATE()) AS 'Hour',
    DATEPART(MINUTE, GETDATE()) AS 'Minute',
    DATENAME(DW, GETDATE()) AS 'DayOfWeek',
    dm.wait_type AS 'WaitType',
    dm.wait_time_ms - ws.wst_WaitTime AS 'WaitTime',
```

```
dm.waiting_tasks_count - ws.wst_WaitingTasks AS 'WaitingTasks',
dm.signal_wait_time_ms - ws.wst_SignalWaitTime AS 'SignalWaitTime'
FROM sys.dm_os_wait_stats dm
INNER JOIN #ws_Capture ws
ON dm.wait_type = ws.wst_WaitType;

-- Clean up the temp table
DROP TABLE #ws_Capture;
```

Using SQL Server Agent to Schedule Measurements

After selecting a capture method, we need to run the capture T-SQL script to fill our `WaitStats` table with wait statistics information. As described in the baseline pitfalls section earlier, it is very important to always perform your measurements at a fixed interval. This makes comparing measurements a lot easier, since you are always comparing the same time segments. The best way to do this is by using a SQL Server Agent job set to a fixed interval. The interval can be set to your choosing—the larger you set the interval, the smaller the number of time segments you can compare against. Setting the interval to be shorter will give you more time segments, but will also mean an increase in data that you need to store. I personally prefer to set my interval to 15 minutes. This gives me enough time segments to compare in most cases.

I won't go into details here about how you can create a SQL Server Agent job to capture wait statistics information, but I do want to point out how my job looks as an example you can use. I usually end up with a SQL Server Agent job with just one T-SQL script step. In this step I copy the capture script, depending on which method I want to use. Figure 4-7 shows a screenshot of my SQL Server Agent job.

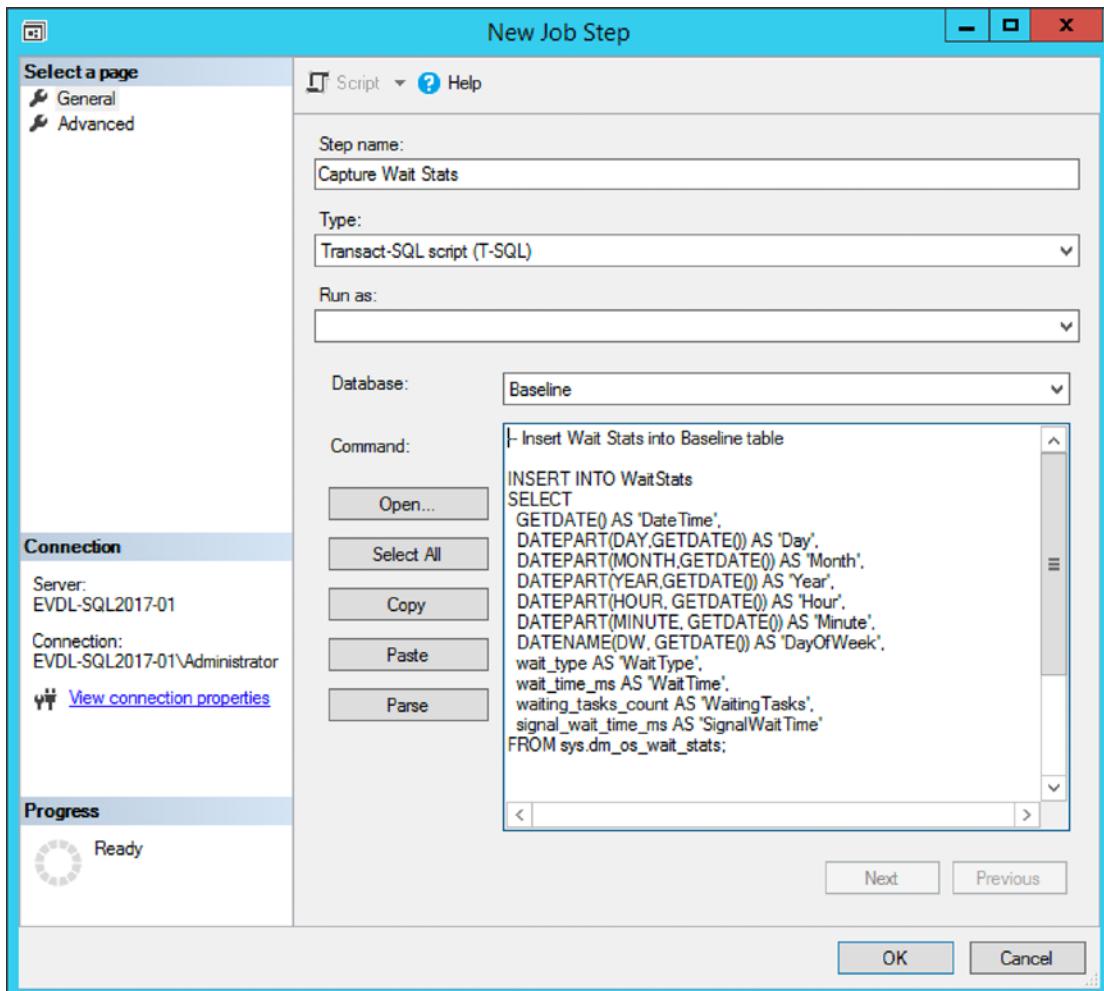


Figure 4-7. Capturing WaitStats SQL Server Agent job step

In this case I used the reset capture method to capture the wait statistics in my `WaitStats` table.

Figure 4-8 shows the schedule I use to capture the wait statistics on a fixed interval. As you can see, I have set it to every 15 minutes, every day.

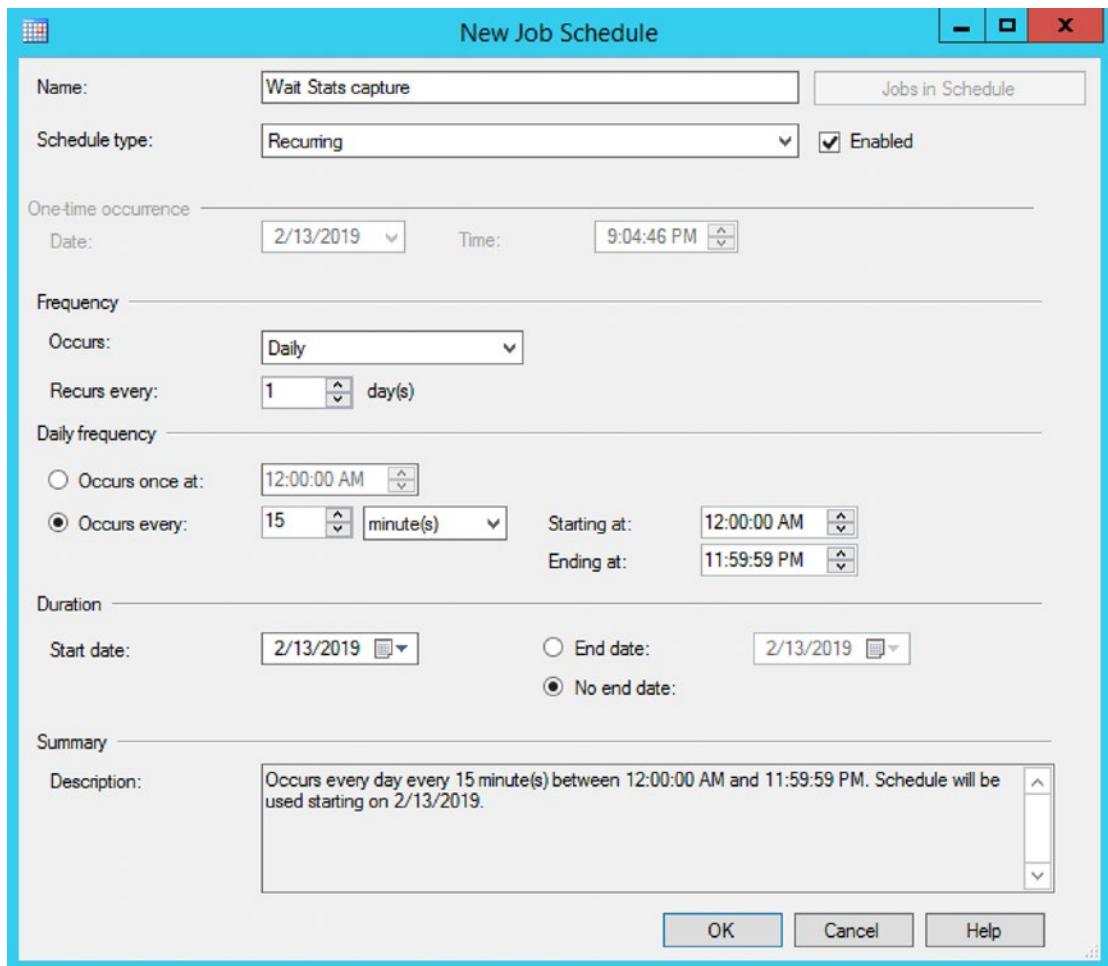


Figure 4-8. SQL Server Agent job schedule

Again, you are free to choose your own capture interval, but make sure to always capture at the same interval length.

After we have created a SQL Server Agent job to gather wait statistics information, we need to let it run for a while. The longer the job runs, the more information we gather, improving the quality of our baselines.

Wait Statistics Baseline Analysis

After letting the SQL Server Agent job collect wait statistics metrics for a while, we are ready to actually create some baselines. The way we do this is by querying the `WaitStats` table we created earlier. I will give you some examples of queries that will create a

baseline you can compare against; these are not the only queries you can run, however, and I encourage you to experiment with different queries to return the information you are most interested in.

Before we get started with building the baseline, I want to return to Figure 2-14 in Chapter 2, “Querying SQL Server Wait Statistics.” In this flowchart I showed you steps you can take to analyze resource waits that occur right now. Since we now have access to a baseline, we can add an extra step to the flowchart. Figure 4-9 shows how to complete the flowchart, including the baseline comparison step.

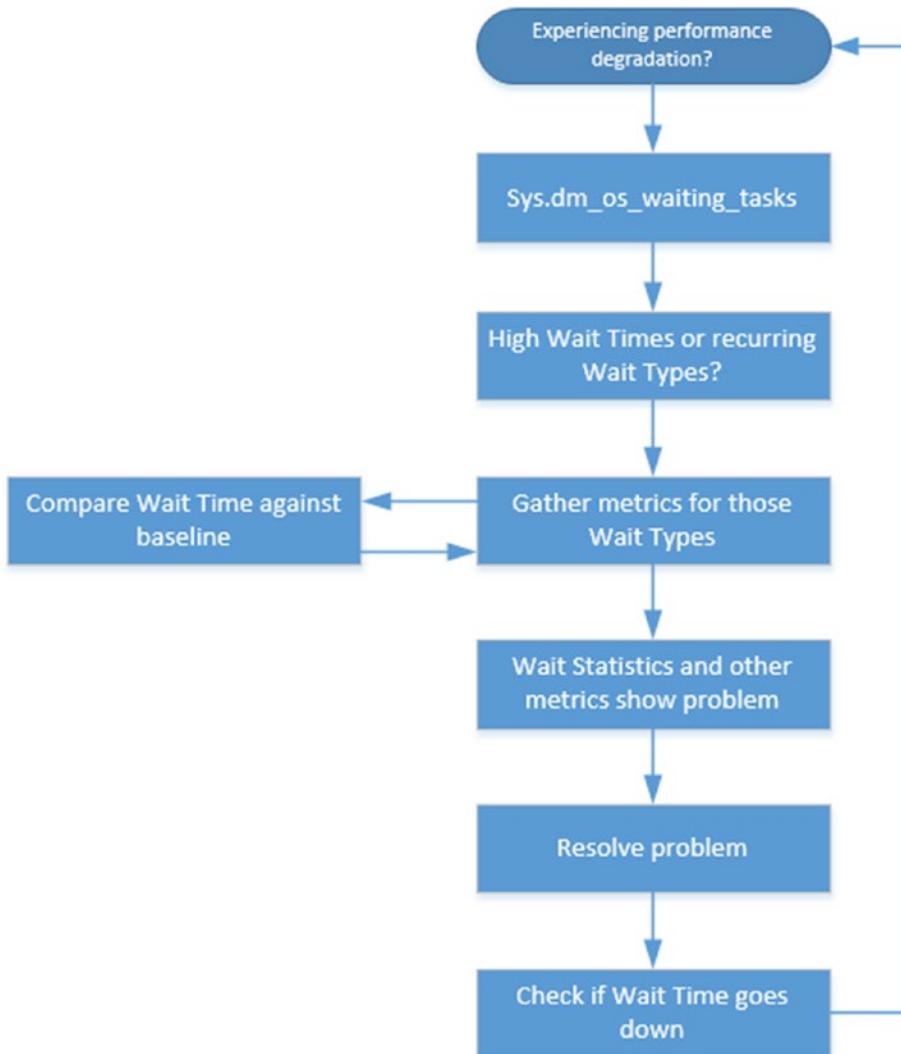


Figure 4-9. Complete wait statistics performance-analysis flowchart

The baselines you create are an extra input to the metrics you gather when looking at a performance problem. They are a very valuable input because they will show you information about the time the problem didn't exist.

Let's go through an example, using DBA Jim again, where we review all the steps of the flowchart shown in Figure 4-9. In this example I will show you queries that you can use against the `WaitStats` table so as to build a baseline that is useful for the performance-analysis process.

Tuesday, around 9 AM, DBA Jim receives a phone call that the daily reporting against the sales database is a lot slower than normal. The problem started around 8 AM, and users are still experiencing performance problems. The reports are part of a scheduled job that runs every workday, starting at 8 AM.

The first thing Jim does is query the `sys.dm_os_waiting_tasks` DMV using the following query:

```
SELECT * FROM sys.dm_os_waiting_tasks
ORDER BY session_id ASC;
```

Jim focuses on user sessions (normally higher than ID 50) but doesn't see any long wait times on any of the user sessions, as shown in Figure 4-10.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
20	0x00000039FA027C28	19	0	6347173	ONDemand_Task_Queue	0x0000003AC3BBEF50
21	0x00000039F152E4E8	27	0	287001609	HADR_NOTIFICATION_DEQUEUE	0x0000003ACB1EEE00
22	0x00000039EE05D0...	31	0	677	SLEEP_TASK	NULL
23	0x00000039EE05C8...	32	0	7008297	BROKER_EVENTHANDLER	NULL
24	0x00000039EB03E4E8	33	0	287001105	BROKER_TRANSMITTER	NULL
25	0x00000039EB03E8C8	34	0	239	HADR_FILESTREAM_IOMGR_IOCOMPLE...	NULL
26	0x00000039FA027848	2060	0	4	PAGEIOLATCH_SH	0x0000003AC4CAEC90
27	0x00000039FA027848	4002	0	10	PAGEIOLATCH_SH	0x0000003AC50AEA00
28	0x00000039FA027848	4519	0	18	PAGEIOLATCH_EX	0x0000003AC3BBEF50

Figure 4-10. `sys.dm_os_waiting_tasks`

After executing the query against the `sys.dm_os_waiting_tasks` DMV multiple times, Jim notices that the wait type `PAGEIOLATCH_SH` is returned every time he queries the DMV. Each time, the wait type is returned with a different session ID but with relatively low wait times.

Jim uses the same T-SQL script to capture wait statistics metrics into the `WaitStats` table, as we discussed earlier in this chapter. Because Jim has access to historic wait statistics information, he decides to create a baseline of the `PAGEIOLATCH_SH` wait times.

The first thing he does is view the PAGEIOLATCH_SH wait times of today, filtered to show measurements captured between 8 and 9 in the morning, using the query shown in Listing 4-5.

Listing 4-5. Show wait times for PAGEIOLATCH_SH between 8 and 9 AM today

```
-- PAGEIOLATCH_SH waits, today between 8 and 9 AM
SELECT
    CONVERT(VARCHAR(5), ws_DateTime, 108) AS 'Time',
    ws_WaitTime AS 'Wait Time'
FROM WaitStats
WHERE ws_WaitType = 'PAGEIOLATCH_SH'
AND (ws_Hour >= 8 AND ws_Hour < 9)
AND CONVERT(VARCHAR(5), ws_DateTime, 105) = CONVERT(VARCHAR(5), GETDATE(), 105)
```

The query returned the results shown in Figure 4-11.

	Time	Wait Time
1	08:00	1528749
2	08:15	1828749
3	08:30	1658974
4	08:45	1698547

Figure 4-11. PAGEIOLATCH_SH wait times of today

Now that Jim has the wait times for today of the PAGEIOLATCH_SH wait type, the next step is to create a baseline from the historic measurements of the PAGEIOLATCH_SH wait type so he can compare today's measurements against the baseline. Jim uses the query shown in Listing 4-6 to build his baseline.

Listing 4-6. PAGEIOLATCH_SH baseline

```
-- Baseline between 8 and 9 on workdays
-- Not including measurements done today
SELECT
    CONVERT(VARCHAR(5), ws_DateTime, 108) AS 'Time',
    AVG(ws_WaitTime) AS 'Baseline'
```

```
FROM WaitStats
WHERE ws_WaitType = 'PAGEIOLATCH_SH'
AND ws_DayOfWeek IN ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
AND (ws_Hour >= 8 AND ws_Hour < 9)
AND CONVERT(VARCHAR(5), ws_DateTime, 105) < CONVERT(VARCHAR(5), GETDATE(), 105)
GROUP BY CONVERT(VARCHAR(5), ws_DateTime, 108);
```

This query builds a baseline with the following characteristics: return the average wait time of PAGEIOLATCH_SH wait type captured on a workday between 8 AM and 9 AM, excluding today. The reason to exclude today is that the measurements that were performed today, during the performance problem, might impact the average. Another suggestion could be to filter only data captured in the last x weeks so as to limit the amount of data that needs to be calculated in the average.

The results of the query shown in Listing 4-6 can be seen in Figure 4-12.

	Time	Baseline
1	08:00	313038
2	08:15	391444
3	08:30	498923
4	08:45	570782

Figure 4-12. PAGEIOLATCH_SH baseline

As you can immediately see when you compare the wait times in Figures 4-11 and 4-12, the measurements done today are a lot higher than those in the historic baseline. To make it a little easier to see the difference, I created a graph of both measurements, as shown in Figure 4-13.

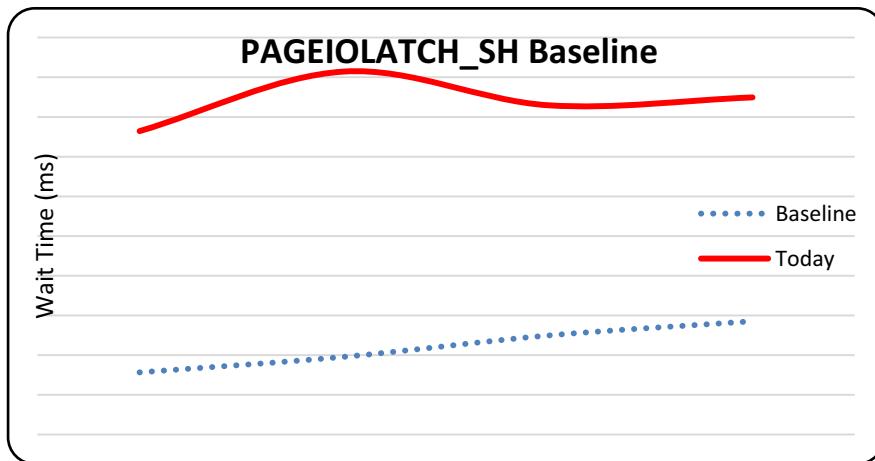


Figure 4-13. Baseline comparison graph for the PAGEIOLATCH_SH wait type

Because there is such a difference in wait times for the PAGEIOLATCH_SH wait type between the baseline and today, Jim believes the PAGEIOLATCH_SH wait type needs further investigation.

We will take a detailed look at the PAGEIOLATCH_SH wait type in Chapter 9, “Latch-Related Wait Types,” but to give you a (very) short explanation, long PAGEIOLATCH_SH waits can indicate storage problems.

To investigate further, Jim starts the Windows Performance Monitor to look at metrics related to the storage subsystem, and in particular the disk latency counters. As you can see in Figure 4-14, the latency on the disk where the database data file resides peaks to very high values, more than 4,000 milliseconds! For SQL Server to perform optimally, the disk latency should be as low as possible, and at least below 20 milliseconds.

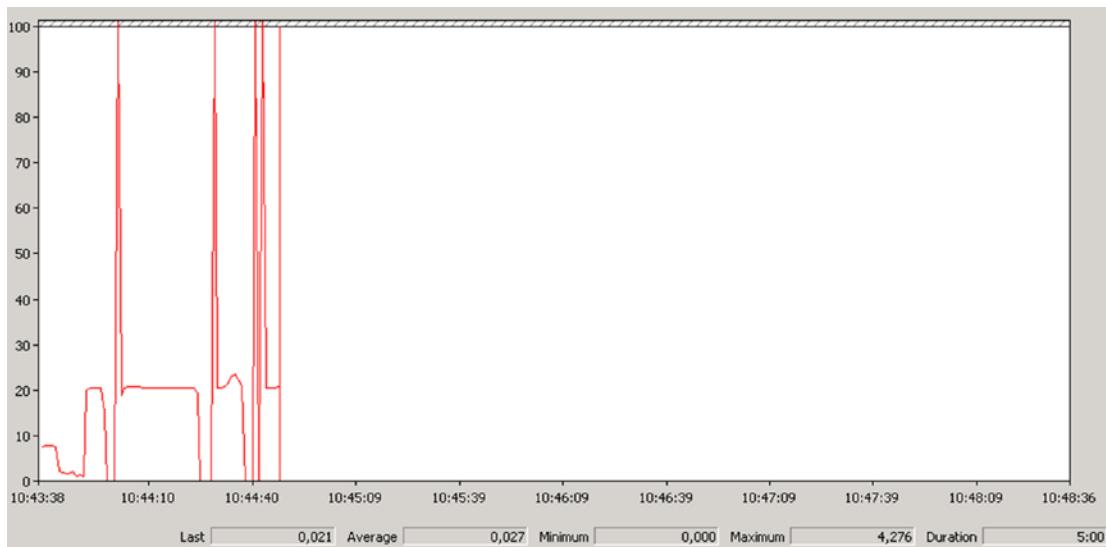


Figure 4-14. Disk-read latency

With both the wait statistics baseline information and the Perfmon metrics, Jim believes the problem is storage related and contacts the storage administrator. The metrics Jim collected also help the storage administrator, since he can compare his storage-related measurements against those Jim supplied. The storage administrator confirms there is a problem related to the disk that contains the sales database, and solves the problem by replacing a faulty disk in the disk array. After the disk gets replaced, the disk latency returns to a 6 milliseconds average, and the high latency peaks disappear. Jim queries the wait times again from the `WaitStats` table after the disk is replaced and notices the wait times for the `PAGEIOLATCH_SH` wait type are close to the baseline values again. Users also inform Jim that the reports are running normally again.

During this example Jim went through all the steps of the wait statistics performance-analysis flowchart shown in Figure 4-9:

1. Users experience performance degradation while running reports.
2. Jim queries the `sys.dm_os_waiting_tasks` DMV to find out if there are high wait times or frequently recurring wait types. The `PAGEIOLATCH_SH` wait type seems to be recurring frequently.
3. Jim gathers metrics by capturing the `PAGEIOLATCH_SH` wait times of today and comparing them to the baseline. He also gathers additional metrics from Perfmon.

4. All the metrics show Jim that the problem is most likely storage related, and Jim contacts the storage administrator.
5. The storage administrator replaces a broken disk in the array. Storage latency values drop to 6 milliseconds.
6. Jim checks the wait times of the PAGEIOLATCH_SH wait type again and confirms that they are close to the baseline values.

Even though this example might seem very simple, it is actually based on a performance problem I encountered in the real world. Using the steps from the wait statistics performance-analysis flowchart combined with the baseline metrics, I was able to identify and solve the problem very quickly.

In the example I showed you the query in Listing 4-6 that creates a baseline for the PAGEIOLATCH_SH wait type. This query is just an example of what you can use against the WaitStats table. You can modify it to suit your own needs; for instance, you can choose to not limit the results for weekdays, and only show average wait times captured on a specific day. Or you could request the actual wait times on a specific date.

If you are capturing wait statistics measurements for a long period of time, it might be a good idea to split the results into multiple tables for easier and faster querying. For instance, you could use the following query to insert all the wait statistics measurements done in March into their own table:

```
SELECT *
INTO WaitStats_March
FROM WaitStats
WHERE ws_Month = 3;
```

This also gives us options to compare specific wait times during different periods of time by joining the different tables together. Figure 4-15 shows the tables of my baseline database that I usually end up with, sorting the data per month.

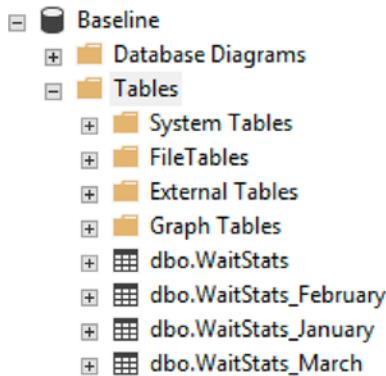


Figure 4-15. Wait statistics measurement split per month

You can decide how you want to split the measurements yourself; maybe you want to store the wait statistics measurements in a separate table for every application version you use, or store all the measurements of a specific wait type in a separate table. The choice is yours.

This chapter hopefully gave you some ideas on how to store wait statistics measurements and create baselines from those measurements. I tried to avoid telling you exactly what to do and how to do it, because I believe one single approach doesn't work for everybody. You will need to write and adjust your own queries to create the baselines you are interested in, but I hope this chapter showed you the foundations to further build upon.

Summary

In this chapter we took a close look at baselines from both a theoretical and a practical point of view. Baselines are incredibly important for any type of performance analysis you perform. In the case of wait statistics, baselines are frequently required if you want to troubleshoot SQL Server-related performance problems. Since wait statistics are unique for your system, there is only one method by which to compare wait times—baselines.

I gave you some examples and T-SQL scripts to create your own wait statistics baseline table so you can start capturing wait statistics information right now. We also went through an example of how you can query that baseline information and compare it to actual measurements to troubleshoot a performance-related incident.

PART II

Wait Types

CHAPTER 5

CPU-Related Wait Types

Processors have evolved enormously in the last few decades, and processor manufacturers, like Intel or AMD, manage to build faster processors on a yearly basis. And while the speed of processors is hitting a ceiling, the number of cores manufacturers manage to build inside their processors has only grown. At the time of writing this book, you can buy a single processor with 24 cores inside to power your system. Processors are also one of the more difficult parts of your system to replace. While you can expand your system's memory relatively simply, replacing a processor for one that is faster or has more cores frequently requires you to change your system's motherboard as well due to CPU socket incompatibility. This means we are usually stuck with our processors until we replace the system altogether.

Processors are also very important for SQL Server. Higher processor speeds will accelerate processor-related instructions, and more cores means more schedulers that SQL Server can use to execute requests. But even all these upgrades in speed and cores cannot prevent the fact that we sometimes have to wait on processor resources. In this chapter we will take a look at some of those wait types that have a relation with your system's processor.

CXPACKET

The first CPU-related wait type is also the most common wait type in SQL Server instances that run with the default, out-of-the-box, SQL Server configuration. It is also one of the most misunderstood wait types and sometimes doesn't even need lowering in order to make your queries perform faster; as a matter of fact, lowering CXPACKET wait times can sometimes degrade the performance of your queries! If you are running SQL Server 2016 SP2 or SQL Server 2017, there have been some changes in how to handle CXPACKET waits. We will discuss the impact of the changes, including the new parallelism-related wait type CXCONSUMER, at the end of this section.

What Is the CXPACKET Wait Type?

The CXPACKET wait type occurs whenever a query is being executed in parallel instead of serial. Parallel queries can have a performance advantage compared to serial queries, if the work can be divided among multiple worker threads. The advantage is bigger for queries that are returning large result sets; queries that return only a few rows benefit far less from parallelism, and in many cases parallelism can slow down those queries. This doesn't mean we should turn off parallelism immediately as I have yet to see a true OLTP database where every query only returns a handful of rows. Many systems have to deal with a mixed workload, usually dealing with many short queries but also large, longer-running, reporting queries.

Parallel queries will use multiple worker threads to execute a request. Along with the worker threads that are created to perform the work requested, a parallel query will also use a 0 thread, called the control thread. This 0 thread's task is to coordinate the work of the other worker threads. While the 0 thread is waiting for the other worker threads to finish the work they were assigned to perform, it will record wait times of the CXPACKET wait type. To understand this relation a little bit better, take a look at Figure 5-1.



Figure 5-1. Parallel query threading

As soon as the SQL Server Query Optimizer decides on an execution plan that uses parallelism, you will see CXPACKET waits occur. This can be completely normal and is nothing to worry about if you are expecting your queries to run in parallel and they are performing as expected. In those cases you can ignore long wait times on the CXPACKET

wait type. There are, however, cases where you don't want to use parallelism, or when parallelism is negatively impacting the performance of your queries because of skewed workloads.

Because the CXPACKET wait type is directly related to the parallelism settings of your SQL Server instance, we can influence it relatively easily by adjusting these settings. We can find the parallelism settings in the **Server Properties > Advanced > Parallelism** section of your SQL Server instance, as shown in Figure 5-2.

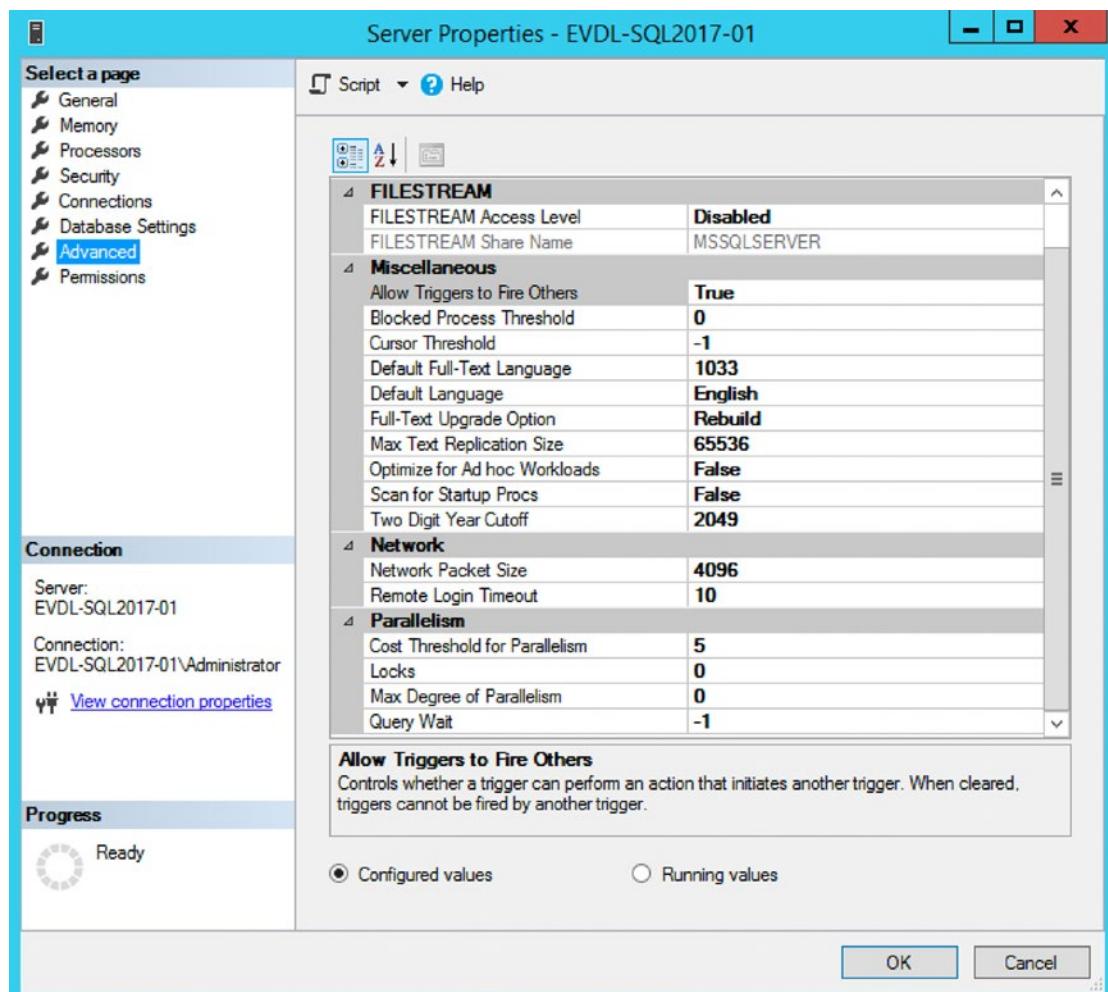


Figure 5-2. Parallelism configuration

Of these settings, the **Cost Threshold for Parallelism** and **Max Degree of Parallelism** settings impact parallel queries the most.

The **Cost Threshold for Parallelism** setting configures the cost threshold of when a query will be considered to be run in parallel by the Query Optimizer. If a serial query has a cost higher than the value configured in the **Cost Threshold of Parallelism**, the Query Optimizer might decide to generate a parallel plan instead of a serial one. By default, the setting has a value of 5 and can be configured to have a value between 0 and 32,767.

The **Max Degree of Parallelism** setting configures the number of schedulers used when executing a parallel plan. By default, this setting is configured to be 0, which means all available schedulers can be used when a parallel plan is executed.

If you are running SQL Server 2016 or higher, you are also able to configure the parallelism settings on a database level through database scoped configuration items, as shown in Figure 5-3.

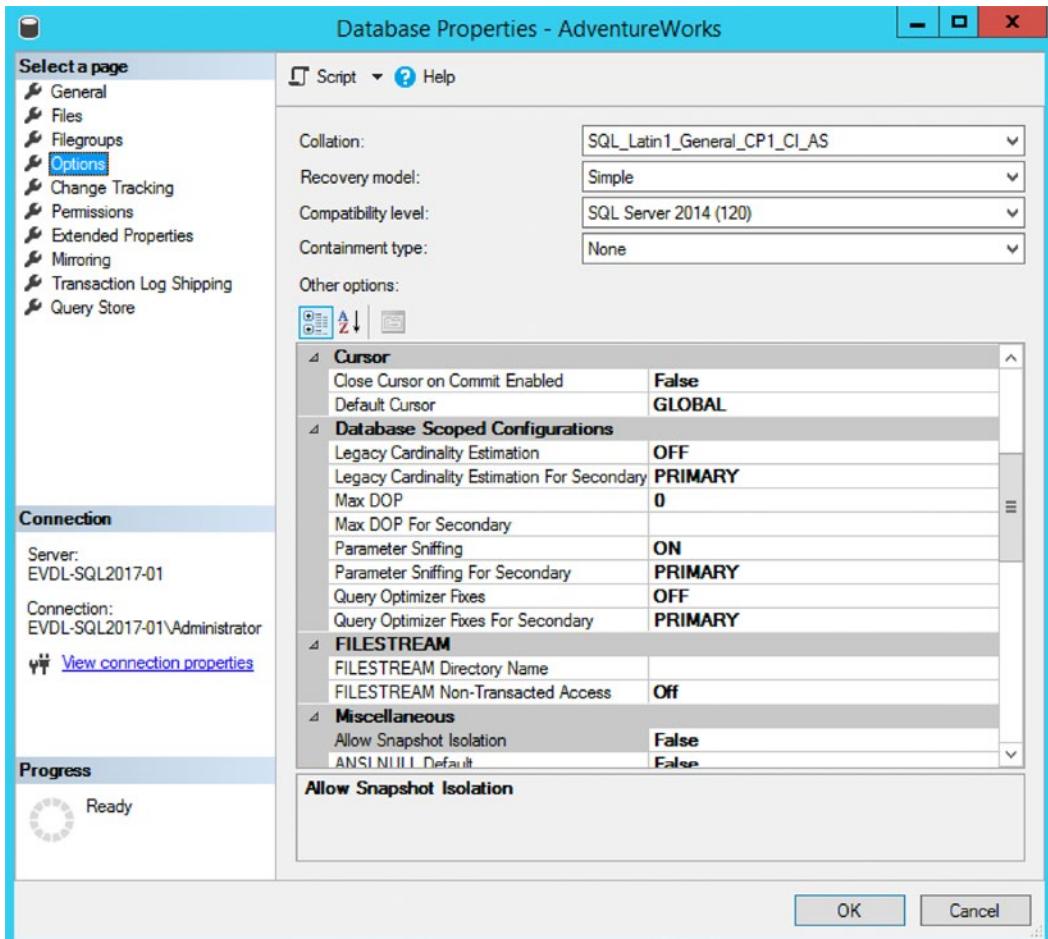


Figure 5-3. Database scoped parallelism configuration

The introduction of the ability to configure setting like parallelism on a per-database level instead of the entire SQL Server instance is a very welcome change. Consider, for instance, that you have multiple databases inside the same SQL Server instance. Ideally each of those databases will use configuration settings that are perfected for their query workload. Before SQL Server 2016, we weren't able to configure that on a per-database level, so generally you would stick with a form of a best-practice or generalized configuration values. Now that database scoped configuration is possible, it is very possible to configure the optimal setting for each individual database.

With the ability to add database scoped configuration values for parallelism settings, there is a difference in how the SQL Server engine processes these configurations:

- The database scoped configuration setting will overwrite the current instance setting only if the database scoped setting is set to a non-default value;
- If the database scoped configuration setting is set to its default value, the instance-wide configuration setting will be used.

As an example, if the Max Degree of Parallelism setting is configured to be 4 on the instance level and 0 (default) on the database level, queries that can be executed in parallel can use four schedulers. If the database scoped setting is changed to be a value of 2, queries executed against the database can use a maximum of two schedulers overwriting the instance setting of 4.

Lowering CXPACKET Wait Time by Tuning the Parallelism Configuration

There are various methods you can use to lower CXPACKET wait times, but before you go and use them you have to be sure that CXPACKET waits are actually causing you problems. Like I said earlier, CXPACKET waits are completely normal whenever you have parallelism enabled for your SQL Server instance. One solution I read frequently on Internet forums is to disable parallelism by setting the **Max Degree of Parallelism** option to a value of 1. In most cases this is not a good idea. Disabling parallelism will make the CXPACKET waits go away completely, but some of your queries might be performing a lot worse since they cannot be run in parallel anymore.

A better approach to lowering CXPACKET waits is to tune the **Cost Threshold for Parallelism** and **Max Degree of Parallelism** options so they match with your workload. This way you can make sure only the queries that benefit the most from parallelism will be run in parallel. A way to find this parallelism sweet spot is by comparing the runtime of a query when it ran serially vs. in parallel. You should generally focus on queries that access a lot of information and have a longer runtime in general, as those will be the queries that benefit the most from parallelism.

Consider this example where we have a query against the AdventureWorks database that requests information from the Sales.SalesOrderDetail table:

```
SELECT *
FROM Sales.SalesOrderDetail
ORDER BY CarrierTrackingNumber DESC;
```

We can check if this query would be a candidate to be run in parallel by checking the estimated cost of the query. To view this information we need to take a look at the estimated execution plan for if the query were to be run serially. To make sure the query runs serially we must add the query option MAXDOP 1. We are also interested in the runtime of the query, so we add the SET STATISTICS TIME ON option to the query:

```
SET STATISTICS TIME ON

SELECT *
FROM Sales.SalesOrderDetail
ORDER BY CarrierTrackingNumber DESC
OPTION (MAXDOP 1);

SET STATISTICS TIME OFF
```

Figure 5-4 shows the estimated cost of the query when run serially.

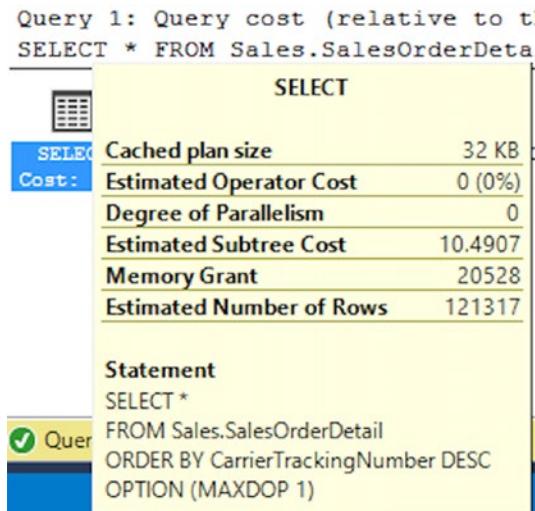


Figure 5-4. Estimated cost of the query with MAXDOP 1

In this case the estimated cost is 10.4907 on my test SQL Server. When I executed the query, the execution time was 2256 milliseconds on my system.

Because the Cost Threshold for Parallelism is still configured on the default value of 5 on my test server, I am pretty sure the query would be run in parallel if I were to remove the MAXDOP query hint.

Figure 5-5 shows the actual execution plan after running the query without the MAXDOP 1 option.

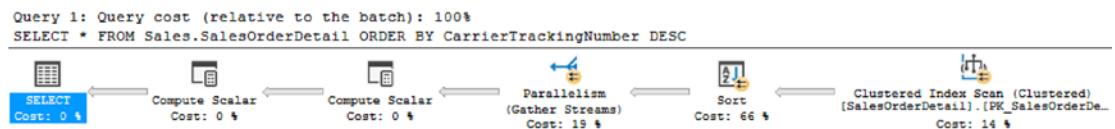


Figure 5-5. Actual execution plan without MAXDOP 1 option

As you can see, the query ran using parallelism, just as we expected, since the estimated cost was higher than the value configured in the Cost Threshold for Parallelism option. The execution time of the query with parallelism was 1959 milliseconds. If we take a look at the properties of the SELECT operation in the actual execution plan, we can view some additional information, as shown in Figure 5-6.

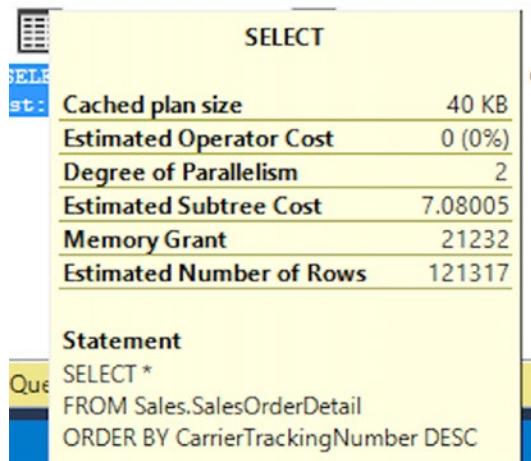


Figure 5-6. *SELECT operation properties*

The properties of the SELECT operation show us that the query was executed using two threads. The estimated cost went down to 7.08005.

Even though the estimated cost went down, the improvement in execution time is pretty small for this query. We could change the Cost Threshold for Parallelism value to a higher number than the default of 5. This way we are making sure relatively small queries like the one in this example don't use parallelism but that heavy reporting queries do.

Another setting to keep in mind is the Max Degree of Parallelism option. When it is set at its default of 0 all available schedulers can be used when a query runs in parallel. Using more schedulers doesn't necessarily mean the query executes faster though. The benefits of using more schedulers slowly get smaller after using more than 8. Microsoft recommends the following configuration in KB2806535:

- For servers with more than eight cores, set the Max Degree of Parallelism option to 8.
- For servers with less than eight cores, set the Max Degree of Parallelism option to 0 or to the number of cores in your server.

This is a general recommendation, and your mileage may vary. The setting of both the Cost Threshold for Parallelism and Max Degree of Parallelism options highly depends on the workload of your system and requires careful testing to find out what works for you and what doesn't. They will impact your CXPACKET wait time though,

so compare your CXPACKET wait times against a baseline after changing the Cost Threshold for Parallelism or Max Degree of Parallelism options to measure the impact of the change.

Lowering CXPACKET Wait Time by Resolving Skewed Workloads

A skewed workload means that all of the worker threads do not receive the same amount of work to perform. This is not an optimal situation, because if one worker thread has to do most of the work while another only a little bit, Thread 0 still has to wait for the longest-running worker thread to complete, logging CXPACKET waits as the time it is waiting. Figure 5-7 shows an abstract example of a skewed workload.

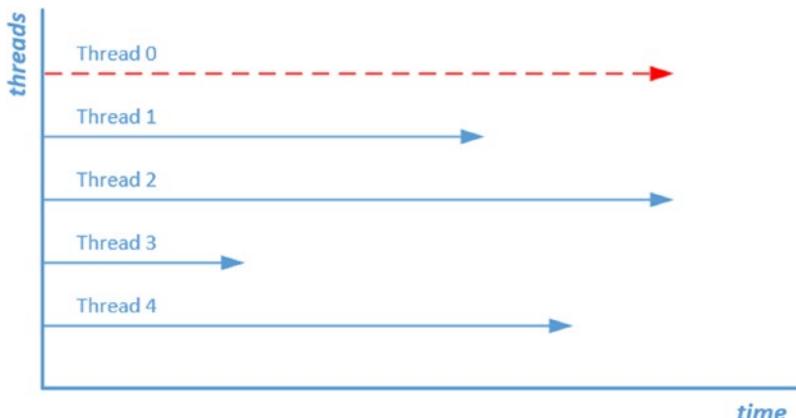


Figure 5-7. Skewed parallel query threading

If we could give some of Thread 2's work to Thread 3, the query would probably perform faster, resulting in lower CXPACKET wait times.

We can view the thread distribution in the actual number of rows property of the parallel operation in the actual execution plan. Figure 5-8 shows the properties of a clustered index scan that has been performed using parallelism. The operation occurred in the example query we used in the previous section against the Sales.SalesOrderDetail table.

Properties	
Clustered Index Scan (Clustered)	
	A Z
Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batches	0
Actual Number of Rows	121317
Thread 0	0
Thread 1	60590
Thread 2	60727
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	

Figure 5-8. Parallel thread distribution

In this example we see that the clustered index scan returned 121.317 rows that were distributed among two threads (notice that Thread 0, the coordination thread, doesn't process any rows). The distribution of the number of rows is relatively even in this case, so we probably aren't running into a skewed workload problem.

Skewed workloads are often caused by outdated statistics. If the Query Optimizer believes there are fewer (or more rows) in the table than there actually are, it can distribute the work unevenly across the threads. Make sure to regularly perform maintenance on your statistics to prevent skewed workloads.

Introduction of the CXCONSUMER Wait Type in SQL Server 2016 SP2 and 2017 CU3

In the release of SQL Server 2017 CU3 (and later SQL Server 2016 SP2), Microsoft pushed a change in how parallelism waits are recorded. The main goal of the development team was to make parallelism wait more actionable. Something that, as you read in the preceding sections, is more than welcome since it is very difficult to determine when parallelism waits are causing issues in your query's performance.

As we described earlier, parallelism consists of two parts: **producers** and **consumers**. The easiest way is to think of the 0 thread we introduced earlier to be a producer. It is the job of the 0 thread to distribute work to the available parallel worker threads. Those worker threads are named consumers and perform the actual work the producers send to them.

Before SQL Server 2017 CU3 and SQL Server 2016 SP2, there is no way to distinguish if, for instance, consumers are spending time waiting on producers to send work to them. Everything is recorded as CXPACKET wait time internally. With the changes in SQL Server 2017 CU3 and SQL Server 2016 SP2, the development team split up the wait times for parallelism into two different categories: CXPACKET and CXCONSUMER. With this change the meaning of those two wait types also changed a bit compared to earlier SQL Server releases.

CXCONSUMER waits can occur whenever a consumer thread is waiting for producer to send rows. This is more or less normal behavior and can in most cases be safely ignored when looking at wait statistics information.

CXPACKET waits are now recorded without the CXCONSUMER wait time, meaning that seeing CXPACKET wait times not only indicate parallelism occurring but also that high wait times indicate a clearer issue regarding parallelism operations (for instance, threads are running into issues with required buffer or thread synchronization). Effectively this means that if you are running SQL Server 2017 CU3 or SQL Server SP2 or higher, seeing CXPACKET waits more clearly indicate parallelism issues than in lower SQL Server versions, thus making the wait type more actionable as the development team intended. The advice in dealing with high parallelism wait times described earlier in this chapter is still valid, though it now has a more direct impact on CXPACKET wait times.

CXPACKET Summary

The CXPACKET wait type is directly related to the usage of parallelism during query execution. If you allow queries to be run using parallelism you will always see CXPACKET waits. Normally this is nothing to worry about, so avoid the knee-jerk reaction to turn off parallelism completely. Instead, focus on tuning the Max Degree of Parallelism and Cost Threshold for Parallelism options so that the thresholds are high enough that your large queries can benefit from using parallelism but your small queries do not experience a negative impact. Also, avoid skewed workloads by making sure your statistics are up-to-date.

If you are running SQL Server 2017 CU3 or SQL Server 2016 SP2 (or higher), the CXPACKET wait time meaning has changed a bit resulting in that CXPACKET waits are far more likely to indicate a parallelism issue occurring than in SQL Server versions that are lower than those mentioned.

SOS_SCHEDULER_YIELD

Just like CXPACKET, SOS_SCHEDULER_YIELD is a wait type that will frequently show up in the top 10 of total wait time on your system. And just like the CXPACKET wait types, SOS_SCHEDULER_YIELD wait times do not necessarily indicate that there is a problem with your SQL Server instance. SOS_SCHEDULER_YIELD waits occur as soon as you start running queries on your SQL Server instance, and they are closely related to SQL Server scheduling.

What Is the SOS_SCHEDULER_YIELD Wait Type?

Before we can answer what the SOS_SCHEDULER_YIELD wait type means, we have to go back to Chapter 1, “Wait Statistics Internals,” in this book, where we discussed SQL Server scheduling. Remember that the SQLOS uses its own cooperative non-preemptive scheduling model to make sure Microsoft Windows processes do not interrupt SQL Server’s own processes? The SOS_SCHEDULER_YIELD wait type has a direct relation with the SQLOS’s cooperative, non-preemptive scheduling model. To make it a little bit easier to understand, I have included Figure 5-9, which should be familiar to you as it represents a scheduler that we discussed in Chapter 1.

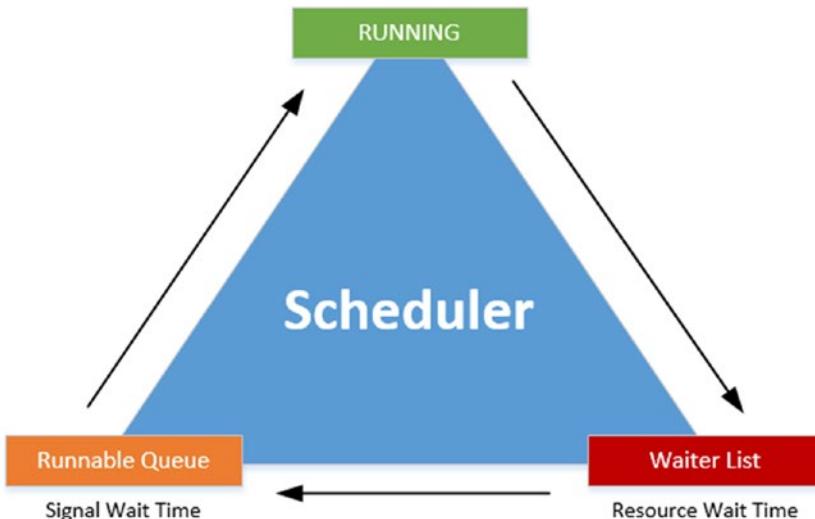


Figure 5-9. Scheduler and its phases and queues

If you remember from Chapter 1, “Wait Statistics Internals,” worker threads move through the different phases and queues in a fixed order. Generally, a worker thread starts on the Waiter List while it waits for resources, it then moves to the Runnable Queue waiting for its turn to be run on the processor, and finally receives processor time to execute its request, receiving the “RUNNING” state. If the worker thread needs additional resources while it is in the “RUNNING” state, the worker thread moves back to the Waiter List and starts a new trip through the different queues and phases.

There is one exception to this behavior and it occurs when a worker thread is in the “RUNNING” state and doesn’t need additional resources to complete its work. If the SQLOS let a worker thread run on the processor for as long as it didn’t need any additional resources, the processor could be “hijacked” by one single worker thread for an infinite amount of time. To make sure a situation like this cannot occur, the scheduler gives every worker thread a specific slice of time in which they need to perform their work. We call this slice of time a quantum, and it is a fixed, unchangeable, 4 milliseconds. If a worker thread spends its quantum it has to yield the processor, and it then moves back to the bottom of the Runnable Queue. It will skip the Waiter List because the worker thread doesn’t need additional resources. While the worker thread is waiting to move back to the processor again, the `SOS_SCHEDULER_YIELD` wait type is recorded. Figure 5-10 shows this behavior.

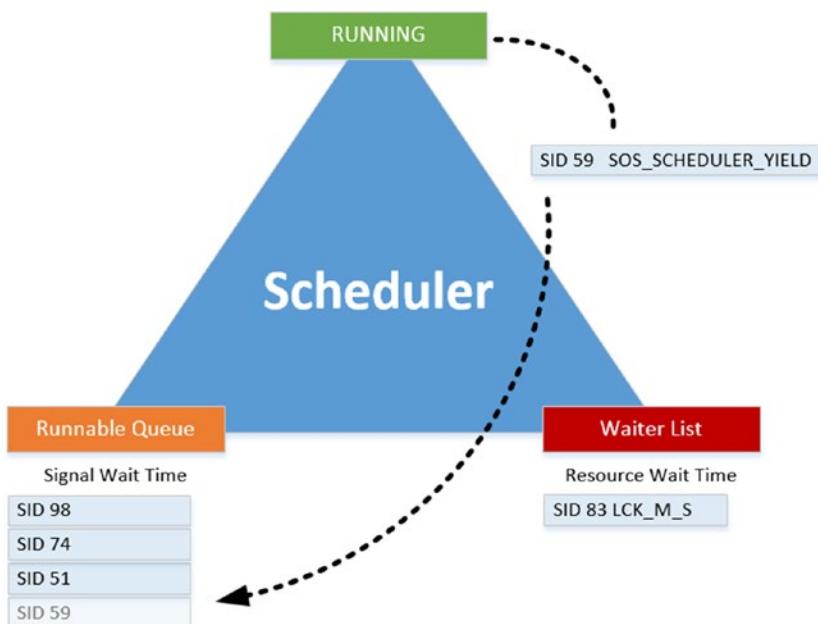


Figure 5-10. Worker thread voluntarily yielding the processor

As you can probably figure out, worker threads are voluntarily yielding all the time, especially on long-running queries where there is no need for additional resources. But keep in mind that wait times for the SOS_SCHEDULER_YIELD wait type will only be logged if the worker thread actually had to wait in the Runnable Queue. If there is no other worker thread in front of the yielding worker thread, it will move directly back to the processor without waiting (it will still move through the Runnable Queue though). To show you an example of this, I executed the following queries against the AdventureWorks database on my test SQL Server, where there is no concurrency whatsoever:

```
-- Clear Wait Stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- Simple select
SELECT *
FROM Sales.SalesOrderDetail
ORDER BY CarrierTrackingNumber DESC;

-- Check for SOS_SCHEDULER_YIELD waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'SOS_SCHEDULER_YIELD';
```

Figure 5-11 shows the results of this query against the sys.dm_os_wait_stats DMV.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SOS_SCHEDULER_YIELD	30	0	0	0

Figure 5-11. SOS_SCHEDULER_YIELD waits

As you can see in Figure 5-11, the query against the AdventureWorks database encountered the SOS_SCHEDULER_YIELD wait type 30 times during execution. It didn't have to spend any time waiting for another worker thread in the Runnable Queue since this was the only query running at the time. If it had spent any time waiting for another worker thread, the wait_time_ms column would have returned a value higher than 0.

As I said at the start of this section, the SOS_SCHEDULER_YIELD wait type is generally not a cause for concern. If, however, the wait times are significantly higher than those in your baseline, it can be a reason to perform some additional research. There are basically three situations you can encounter when dealing with SOS_SCHEDULER_YIELD waits, as shown in Figure 5-12.

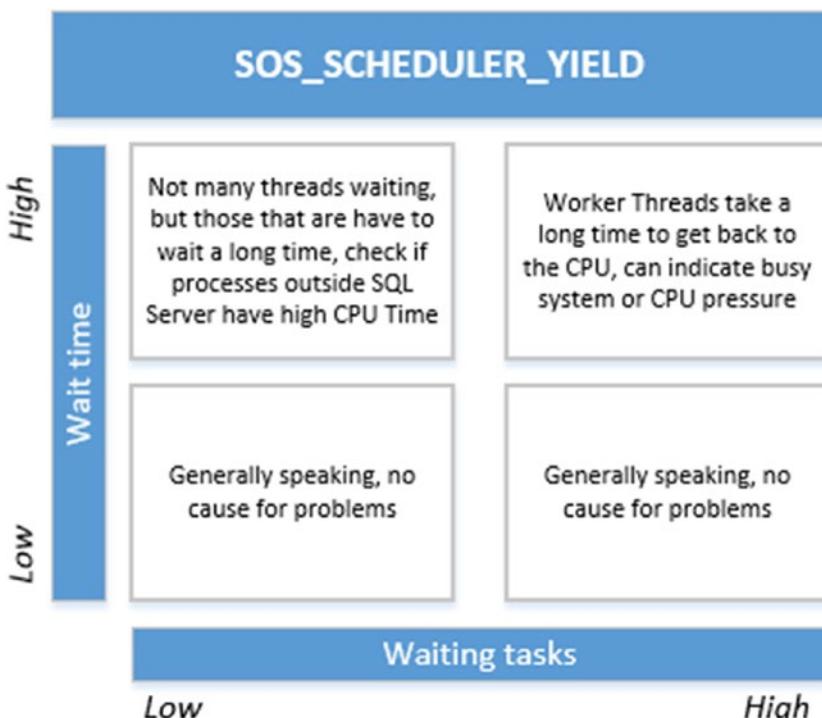


Figure 5-12. SOS_SCHEDULER_YIELD situations

Let's take a look at how we can analyze and resolve the SQL Server CPU pressure problem.

Lowering SOS_SCHEDULER_YIELD Waits

If you are experiencing higher than normal SOS_SCHEDULER_YIELD wait times and a large number of waits, you could, potentially, have a CPU-related problem on your system. To lower the SOS_SCHEDULER_YIELD waits, we are going to focus on the top-right section of Figure 5-12, where there are a large amount of waiting tasks and high wait times.

If you are experiencing high wait times for the SOS_SCHEDULER_YIELD wait type together with a large amount of waiting tasks, you can assume you have a very busy SQL Server instance. Worker threads will yield, but it will take them a long time to get back on the processor again because there are many other threads waiting in the Runnable Queue. As we discussed earlier in Chapter 1, “Wait Statistics Internals,” the Runnable Queue is a first-in first-out list, meaning that the more worker threads that are waiting inside the Runnable Queue, the longer it takes for worker threads to move through it. You will usually see a high CPU usage on the system by the SQL Server process.

To show you an example of this problem, we will use the Ostress utility to execute a specific query simultaneously from a number of threads. The Ostress utility is part of the RML utilities for SQL Server, which you can download here: <https://support.microsoft.com/en-us/kb/944837>.

The first thing we are going to do is save the following query as C:\sos_scheduler_yield.sql on the test server:

```
WHILE (1=1)
BEGIN
    SELECT COUNT(*)
    FROM Sales.SalesOrderDetail
    WHERE SalesOrderID BETWEEN 45125 AND 54185
END;
```

This query will count the number of rows between two SalesOrderIDs in the Sales.SalesOrderDetail table of the AdventureWorks database. It will do this in an endless loop.

After saving the query we start the Ostress utility using the following command:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -i"C:\sos_scheduler_yield.sql" -n20 -r1 -q
```

This starts the Ostress utility, which connects to the AdventureWorks database and executes the sos_scheduler_yield.sql script using 20 threads.

As soon as we start Ostress, the CPU of the test SQL Server hits 100%, as shown in Figure 5-13.

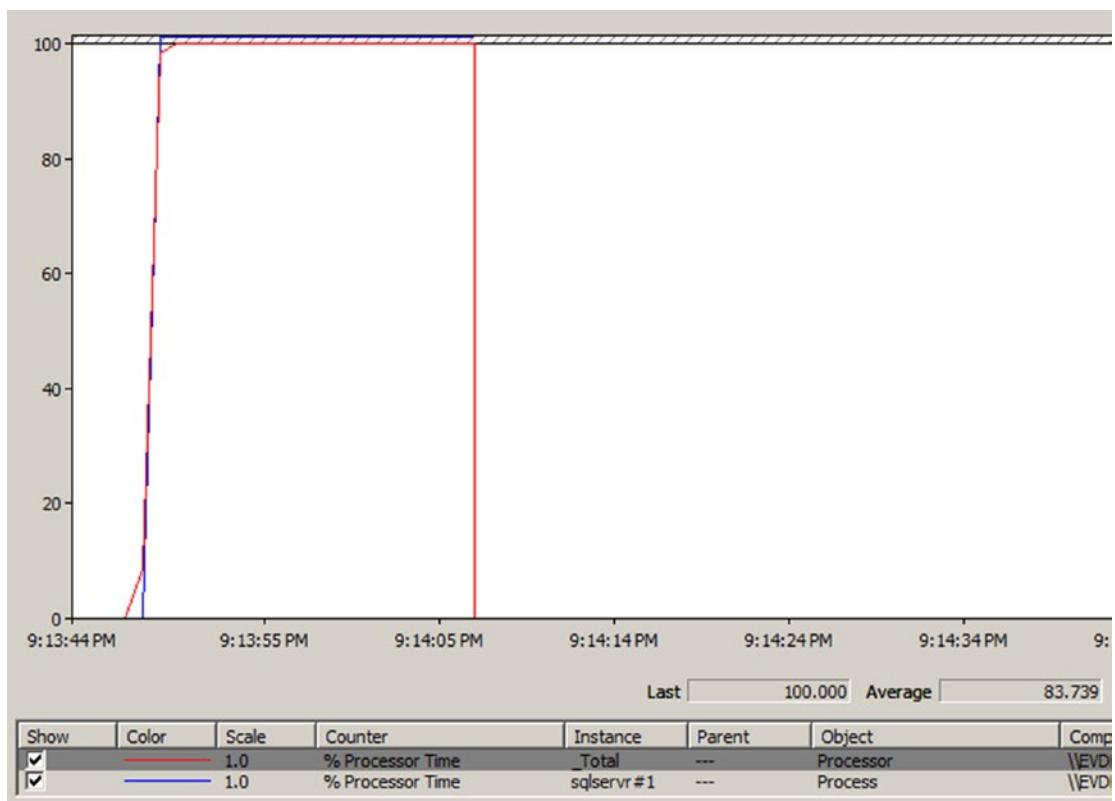


Figure 5-13. Impact of OStress on the CPU

As you can see in Figure 5-13, the CPU load is generated from the sqlservr#1 process, which happens to be the SQL Server instance we are running the OStress query against.

If we were to query the sys.dm_os_waiting_tasks DMV to check if the SOS_SCHEDULER_YIELD wait type is responsible for the CPU usage, we would be in for a surprise, as you can see in Figure 5-14.

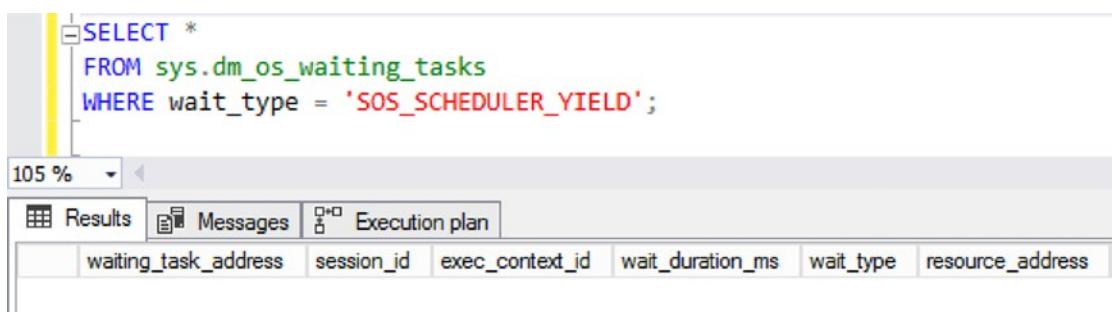


Figure 5-14. No SOS_SCHEDULER_YIELD waits occurring

This is the tricky part of the SOS_SCHEDULER_YIELD wait type, as it frequently won't get returned by the sys.dm_os_waiting_tasks DMV—another reason to capture and use that wait statistics baseline!

To show that the high CPU usage is related to the SOS_SCHEDULER_YIELD wait type, we have to take a look at the cumulative wait statistics DMV, sys.dm_os_wait_stats. We can use the following query to show the top five wait types ordered by wait time while we run the Ostress utility (we can reset the DMV before starting the Ostress utility to keep the numbers small):

```
SELECT TOP 5 *
FROM sys.dm_os_wait_stats
ORDER by wait_time_ms DESC;
```

The results of this query are shown in Figure 5-15.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SOS_SCHEDULER_YIELD	6867	105236	33	108219
2	XE_TIMER_EVENT	4	15001	5001	15001
3	REQUEST_FOR_DEADLOCK_SEARCH	43	15000	5000	15000
4	LAZYWRITER_SLEEP	14	14136	1027	124
5	DIRTY_PAGE_POLL	138	13997	104	2

Figure 5-15. Top five wait types during Ostress execution

As you can see, the number one wait type, by far, is SOS_SCHEDULER_YIELD with a pretty high amount of waiting_tasks and total wait_time.

If you were to experience this problem with a production SQL Server instance, the first thing you should focus on are those very small, very quick queries like the ones we executed in this example. Has the volume of those queries increased? Has the number of user connections to the SQL Server executing those queries increased? Those are two quick questions you should ask and check. A sudden growth in transactions or user connections can lead to high SOS_SCHEDULER_YIELD wait times.

Another cause of high SOS_SCHEDULER_YIELD waits, together with very high CPU usage, can be a phenomenon called *spinlock contention*. Spinlocks are defined by Microsoft as “lightweight synchronization primitives which are used to protect access to data structures” and are a very advanced topic. Appendix II, at the back of this book, goes into a little bit more detail about spinlocks for those who are interested in learning more about them.

Very large, very complex queries can also lead to higher SOS_SCHEDULER_YIELD wait times. Try looking for active queries that consume a lot of CPU time and have complex calculations or data-type conversions inside them. One query I use frequently to identify CPU-heavy queries is the one in Listing 5-1.

Listing 5-1. Detect expensive CPU queries

```
SELECT TOP 10
    QText.TEXT AS 'Query',
    QStats.execution_count AS 'Nr of Executions',
    QStats.total_worker_time/1000 AS 'Total CPU Time (ms)',
    QStats.last_worker_time/1000 AS 'Last CPU Time (ms)',
    QStats.last_execution_time AS 'Last Execution',
    QPlan.query_plan AS 'Query Plan'
FROM sys.dm_exec_query_stats QStats
CROSS APPLY sys.dm_exec_sql_text(QStats.sql_handle) QText
CROSS APPLY sys.dm_exec_query_plan(QStats.plan_handle) QPlan
ORDER BY QStats.total_worker_time DESC;
```

The results of the query in Listing 5-1 on my test SQL Server can be seen in Figure 5-16.

	Query	Nr of Executions	Total CPU Time (ms)	Last CPU Time (ms)	Last Execution	Query Plan
1	WHILE (1=1) BEGIN SELECT COUNT(*) F...	355	1222	2	2019-02-17 14:46:03.840	<ShowPlanXML xmlns="http://schemas.mic...
2	SELECT * FROM Sales.SalesOrderDetail ORDER ...	1	415	415	2019-02-17 14:45:43.250	<ShowPlanXML xmlns="http://schemas.mic...
3	SELECT TOP 10 QText.TEXT AS 'Query', QSta...	5	36	9	2019-02-17 14:47:01.740	<ShowPlanXML xmlns="http://schemas.mic...
4	(@results_row_count int,@interval_start_time datetimeo...	1	12	12	2019-02-17 14:47:13.820	<ShowPlanXML xmlns="http://schemas.mic...
5	(@query_id bigint,@plan_id bigint)SELECT p.is_for...	2	2	0	2019-02-17 14:47:13.960	<ShowPlanXML xmlns="http://schemas.mic...
6	IF OBJECT_ID (N'[sys].[database_query_store_options...)	1	2	2	2019-02-17 14:47:07.577	<ShowPlanXML xmlns="http://schemas.mic...
7	(@results_row_count int,@recent_start_time datetimeo...	1	1	1	2019-02-17 14:47:10.613	<ShowPlanXML xmlns="http://schemas.mic...
8	SELECT actual_state,readyonly_reason FROM sys.dat...	2	0	0	2019-02-17 14:47:13.780	<ShowPlanXML xmlns="http://schemas.mic...
9	SELECT TOP 5 * FROM sys.dm_os_wait_stats OR...	1	0	0	2019-02-17 14:46:58.050	<ShowPlanXML xmlns="http://schemas.mic...
10	select * from sys.dm_os_waits WHERE wait_type =...	1	0	0	2019-02-17 14:46:51.707	<ShowPlanXML xmlns="http://schemas.mic...

Figure 5-16. Expensive CPU queries

As you can see, the query we used with the Ostress tool is the query that got executed the most and took the highest total CPU time. This query could be a good starting point for an investigation. Maybe the query can be optimized or rewritten so it doesn't consume as much CPU time.

Another method you can use to identify queries that are expensive CPU wise is the Query Store. The Query Store offers a built-in report called "Top Resource Consuming Queries" that immediately allows you to filter on CPU time, as shown in Figure 5-17.

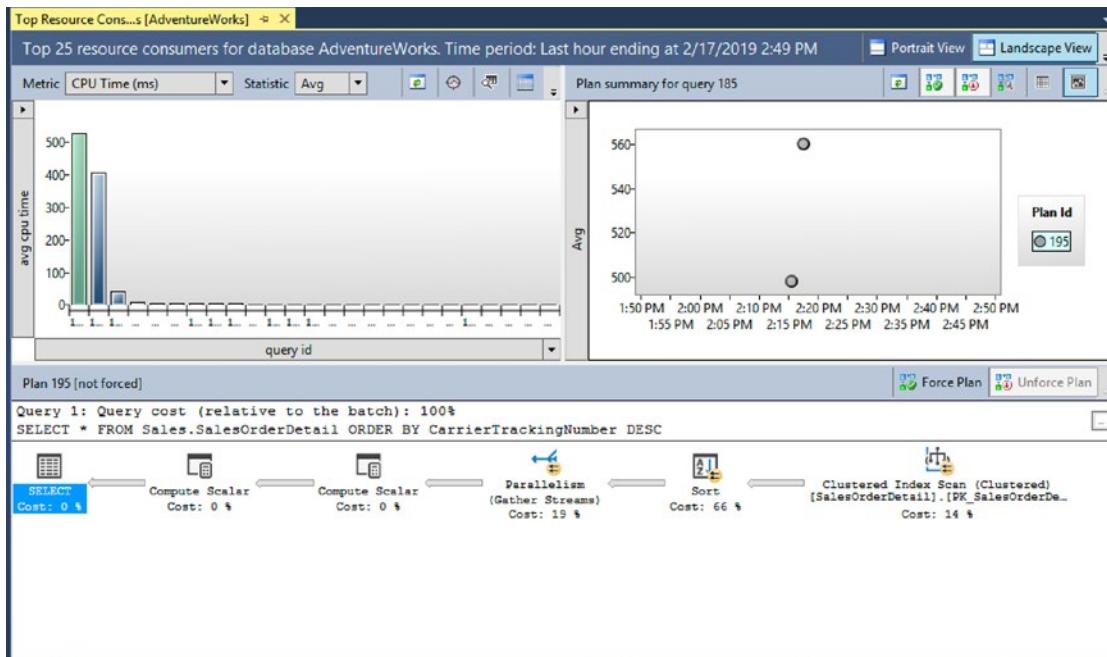


Figure 5-17. Visualizing expensive CPU queries through the Query Store

SOS_SCHEDULER_YIELD Summary

The SOS_SCHEDULER_YIELD wait type will always occur on every SQL Server instance since it is directly related to the scheduling model that SQL Server uses to grant worker threads access to the processor. It can indicate a problem if the total wait time or total amount of waiting tasks suddenly increases compared to your baseline measurements. Most of the time a large increase in SOS_SCHEDULER_YIELD waits also means an increase in the CPU load. This increase can either be caused by the SQL Server process itself or by another process outside of SQL Server that requires a large amount of processor time, limiting the time SQL Server can access the processor. If the SQL Server process is responsible for the increase in CPU load, you should try to correlate the increase in SOS_SCHEDULER_YIELD waits with an increase in user activity. Another option is to query the sys.dm_exec_query_stats DMV, as shown in Listing 5-1, or use the Query Store to find the queries that require the most processor time and focus on optimizing those queries.

THREADPOOL

One of the most notorious wait types is the THREADPOOL wait type. Unlike the CXPACKET and SOS_SCHEDULER_YIELD wait types that occur even if your SQL Server instance isn't experiencing any issues, high THREADPOOL wait times do frequently indicate a performance problem. Just as with the other two CPU-related wait types we discussed in this book, the THREADPOOL wait type is very closely related to the way SQL Server scheduling works.

What Is the THREADPOOL Wait Type?

If you ever see THREADPOOL waits occur on your system with far longer wait times than normal, and your SQL Server is (almost) unresponsive, chances are that you are running into an issue called thread pool starvation. Thread pool starvation occurs when there are no more free worker threads available to process requests. When this situation occurs, tasks that are currently waiting to be assigned to a worker thread will log the THREADPOOL wait type.

SQL Server provides a number of worker threads to the schedulers with which to process requests. The number of worker threads that are available for your system depends on the number of processors and the processor architecture. Table 5-1 shows the maximum number of worker threads available for systems with up to 64 logical CPUs.

Table 5-1. Maximum Number of Worker Threads

CPU Number	32-Bit Architecture	64-Bit Architecture
≤4	256	512
8	288	576
16	352	704
32	480	960
64	736	1472

You can also calculate the maximum number of worker threads available by using these formulas:

- 32-bit systems with less than, or equal to, 4 logical processors:
 - 256 worker threads
- 32-bit system with more than 4 logical processors:
 - $256 + ((\text{number of logical processors} - 4) \times 8)$
- 64-bit system with less than, or equal to, 4 logical processors:
 - 512 worker threads
- 64-bit system with more than 4 logical processors:
 - $512 + ((\text{number of logical processors} - 4) \times 16)$

Even though SQL Server calculates the maximum amount of available worker threads automatically (only once during startup), you can choose to overwrite the default by changing the **Maximum Worker Threads** option inside the Processors properties of your SQL Server instance, as shown in Figure 5-18. By default, the value of the Maximum Worker Threads option will be 0, which means SQL Server will calculate and assign the maximum amount of worker threads available using the preceding formulas.

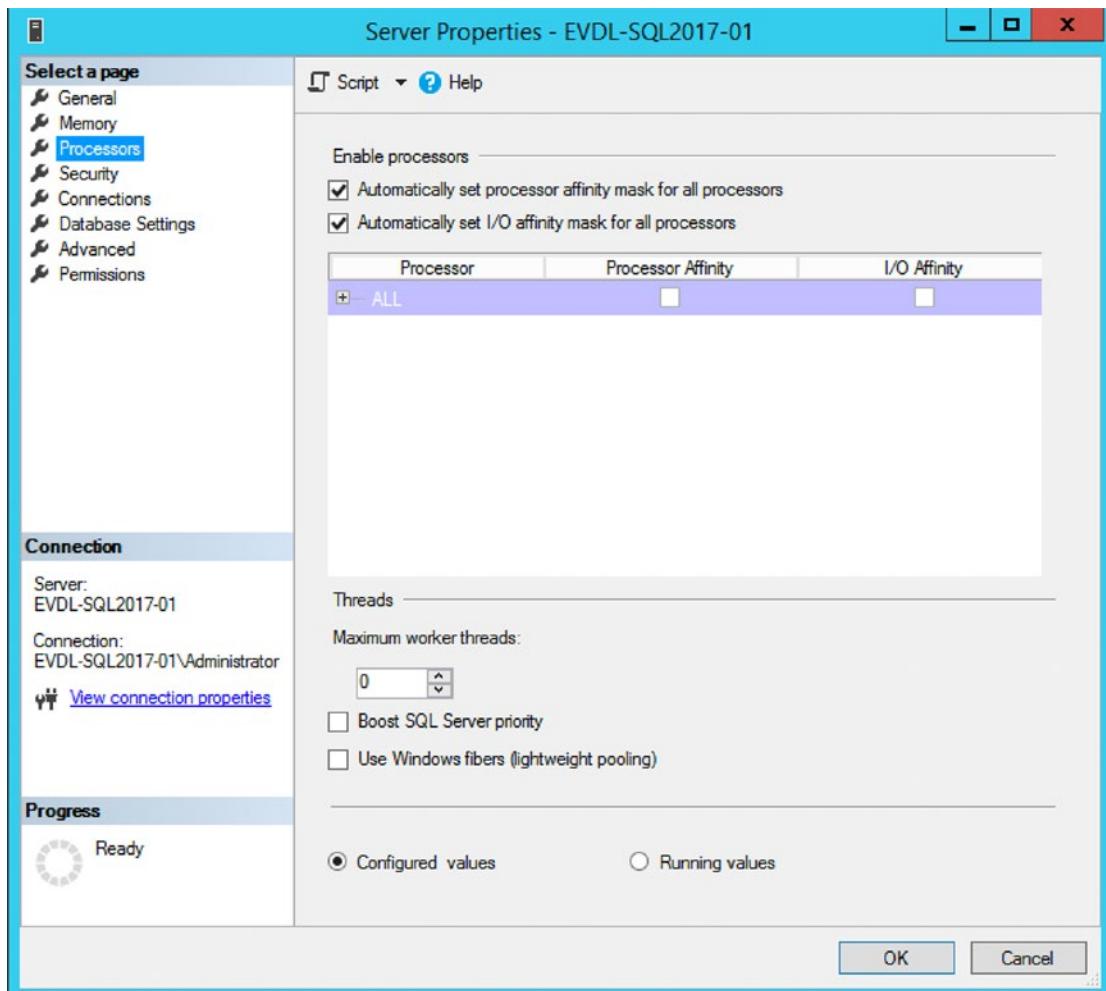


Figure 5-18. Processors configuration of a SQL Server instance

You can also query the number of worker threads assigned to your SQL Server instance by running the following query:

```
SELECT
    max_workers_count
FROM sys.dm_os_sys_info;
```

For my 64-bit test SQL Server that has two logical processors, I have 512 worker threads available, as you can see in Figure 5-19.

	max_workers_count
1	512

Figure 5-19. Amount of worker threads on my test machine

One piece of advice I frequently read on the Internet related to THREADPOOL waits is to change the Maximum Worker Threads option to a value higher than the one your SQL Server instance has by default. I strongly advise against changing this option from its default value. Changing the setting to a higher value than the amount of worker threads you would receive by default can actually degrade the performance of your SQL Server because context-switching occurs far more often. Another reason not to change the setting is that every worker thread requires a bit of memory to operate; for 32-bit systems this is 512 KB per worker thread, and for 64-bit systems it's 2048 KB.

THREADPOOL Example

Let's start with an example of THREADPOOL waits occurring on my test SQL Server instance. Even though I have warned you multiple times already about making sure to not run any of the demo scripts in this book on a production environment, this one deserves a special reminder. Running the demo scripts in this section can cause your SQL Server to become completely unresponsive, not accepting any new connections, and can eventually require a restart of the SQL Server service! Do not run this on a SQL Server that isn't allowed to become unresponsive!

For this example we are going to use the Ostress utility again to simulate concurrency and load against the test SQL Server instance. First, we create another .sql file (`select_rnd.sql`) that holds the following query that we will execute using Ostress:

```
SELECT TOP 1 *
FROM Sales.SalesOrderDetail
ORDER BY NEWID()
OPTION (MAXDOP 1)
```

This query will select one random row from the `Sales.SalesOrderDetail` table in the `AdventureWorks` database. There is a reason I included the query option to serially run this query, and I will explain it later on.

Now, before we launch Ostress to execute the preceding query, we are purposely going to lower the maximum amount of worker threads available on the test SQL Server. To do this we execute this query:

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE
GO
EXEC sp_configure 'max worker threads', 128;
GO
RECONFIGURE
GO
```

This will set the maximum number of worker threads available to 128, the minimum value for a 64-bit SQL Server instance.

Let's fire up Ostress and execute the .sql script we created earlier:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -i"C:\select_rnd.sql" -n150 -r10 -q
```

In this case we will start 150 different threads that will execute the query in the select_rnd.sql file 10 times. The reason for spawning 150 threads is that this value is higher than the maximum amount of worker threads available on the test SQL Server instance, but not so high that we cannot execute queries anymore.

While the script is running, let's take a look at the number of worker threads running and waiting using the sys.dm_osSchedulers DMV:

```
SELECT
    scheduler_id,
    current_tasks_count,
    runnable_tasks_count,
    current_workers_count,
    active_workers_count,
    work_queue_count
FROM sys.dm_osSchedulers
WHERE status = 'VISIBLE ONLINE';
```

The results of this query are shown in Figure 5-20.

	scheduler_id	current_tasks_count	runnable_tasks_count	current_workers_count	active_workers_count	work_queue_count
1	0	88	68	81	80	7
2	1	88	66	80	78	8

Figure 5-20. Tasks and worker threads per scheduler

The most important columns here are the `current_workers_count`, `active_workers_count`, and `work_queue_count` columns. The `current_workers_count` column shows the number of worker threads associated with this scheduler; this number also includes worker threads that are not yet assigned to a task. The `active_workers_count` column returns the number of worker threads that are in the “RUNNING,” “RUNNABLE,” or “SUSPENDED” states. The big difference between the `current_workers_count` and the `active_workers_count` columns is that the `active_workers_count` is the number of worker threads that have been assigned to a task, while the `current_workers_count` returns all the worker threads. The `work_queue_count` column shows us the number of tasks that are currently waiting to get a worker thread assigned to them. If you see values higher than 0 in this column for a longer period of time and for all schedulers, you are experiencing thread pool starvation.

Let’s check the `sys.dm_os_waiting_tasks` DMV for waiting tasks that originate from a user session. Notice that we filter out all the sessions that have a session ID lower than 50, even though I told you to not do this in Chapter 2, “Querying SQL Server Wait Statistics”:

```
SELECT *
FROM sys.dm_os_waiting_tasks
WHERE session_id > 50;
```

If we check the results on the test SQL Server instance, we could conclude that nothing is waiting, as you can see in Figure 5-21. The test SQL Server is responding incredibly slowly though, and querying anything requires multiple seconds.

waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address

Figure 5-21. No tasks are waiting

Let's check the sys.dm_os_waiting_tasks DMV without filtering out session IDs:

```
SELECT *
FROM sys.dm_os_waiting_tasks;
```

As you can see in Figure 5-22, THREADPOOL waits are not logged as user sessions, but actually have an empty session ID. This is the reason I always recommend to not filter the sys.dm_os_waiting_tasks DMV on session ID numbers.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
12	0x00000039E120A4E8	NULL	NULL	1719	THREADPOOL	NULL
13	0x00000039E120A8C8	NULL	NULL	158	THREADPOOL	NULL
14	0x00000039E120AC8	NULL	NULL	30	THREADPOOL	NULL
15	0x00000039E120B088	NULL	NULL	30	THREADPOOL	NULL
16	0x00000039E120B468	NULL	NULL	30	THREADPOOL	NULL
17	0x00000039E120B848	NULL	NULL	30	THREADPOOL	NULL
18	0x00000039E120BC28	NULL	NULL	30	THREADPOOL	NULL
19	0x00000039E124A108	NULL	NULL	30	THREADPOOL	NULL

Figure 5-22. THREADPOOL waits

There are quite a lot of THREADPOOL waits, with various wait times, some running into seconds of wait time. Things can get even worse than this though. Figure 5-23 shows an error I encountered when I tried to connect to my test SQL Server instance while running the Ostress tool.

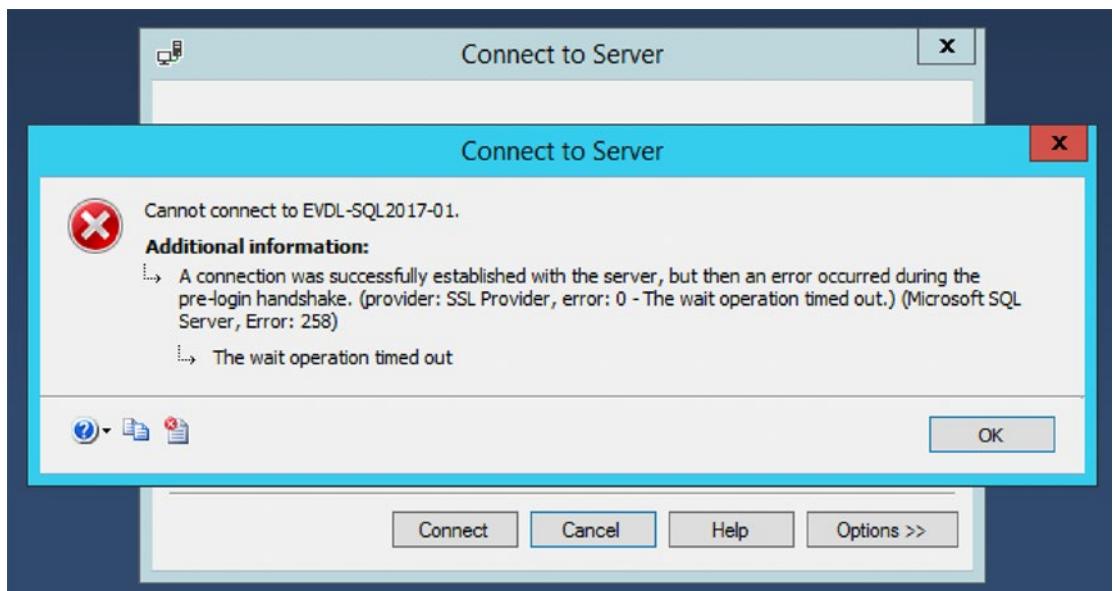


Figure 5-23. Timeouts are occurring and SQL Server is unresponsive

Now that we have seen the kind of problems thread pool starvation can create, let's take a look at how we can lower, or even resolve, THREADPOOL waits.

Gaining Access to Our SQL Server During THREADPOOL Waits

THREADPOOL waits can be very difficult to troubleshoot, mostly because there are many possible reasons why your SQL Server doesn't have any free worker threads available. Also, THREADPOOL waits can completely lock down your SQL Server instance, making connections to it (and troubleshooting it) almost impossible, as you have seen in the earlier example.

The first step you should take to make sure you do not get into a situation where you cannot connect to your SQL Server instance for troubleshooting is to enable the Dedicated Administrator Connection (or DAC). If you remember the section about schedulers in Chapter 1, “Wait Statistics Internals,” you might recall a special type of scheduler reserved for the DAC. This dedicated scheduler, shown in Figure 5-24, is strictly reserved for the DAC and has access to its own worker threads.

	scheduler_address	parent_node_id	scheduler_id	cpu_id	status	is_online	is_idle
1	0x00000039FA180040	0	0	0	VISIBLE ONLINE	1	1
2	0x00000039FA1A0040	0	1	1	VISIBLE ONLINE	1	0
3	0x00000039FA1C0040	0	1048578	0	HIDDEN ONLINE	1	0
4	0x00000039FA800040	64	1048576	0	VISIBLE ONLINE (DAC)	1	1
5	0x00000039F5960040	0	1048579	1	HIDDEN ONLINE	1	1
6	0x00000039F5940040	0	1048580	0	HIDDEN ONLINE	1	1
7	0x00000039F2880040	0	1048581	1	HIDDEN ONLINE	1	1
8	0x00000039F28A0040	0	1048582	0	HIDDEN ONLINE	1	1
9	0x00000039F28E0040	0	1048583	1	HIDDEN ONLINE	1	1

Figure 5-24. Dedicated Administrator Connection scheduler

If you connect through the DAC to your SQL Server instance, your session will be mapped to the DAC scheduler. This makes it possible to connect and execute queries even if all the other schedulers have massive task queues.

You can enable the DAC by executing the following query:

```
sp_configure 'remote admin connections', 1  
GO  
RECONFIGURE  
GO
```

If you want to connect to your SQL Server instance using the DAC you need to add the ADMIN: prefix to the server name you are connecting to, as shown in Figure 5-25. You can only connect using the DAC when you execute a new query from inside SQL Server Management Studio without being connected to the server.

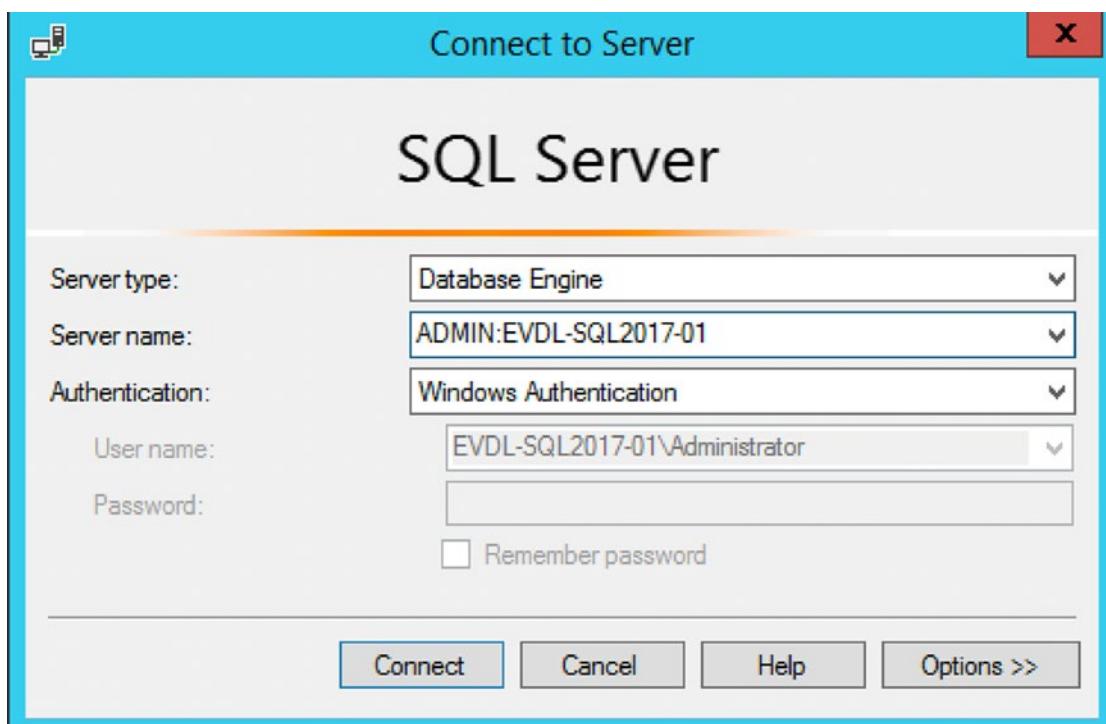


Figure 5-25. Connect using the Dedicated Administrator Connection

Now that you are able to connect to your SQL Server instances using the DAC, you always have a way in, even when the SQL Server instance won't accept any new connections.

With the DAC enabled, let's discuss some common causes for THREADPOOL waits.

Lowering THREADPOOL Waits Caused by Parallelism

One of the most common causes for THREADPOOL waits I encounter is related to the extensive use of parallelism during query execution. During the execution of a parallel query, multiple worker threads are used to perform the work needed. If you left the configuration options related to parallelism—Max Degree of Parallelism and Cost Threshold of Parallelism—at the default values, it might cause more queries to run in parallel than was intended. Depending on how many processors your SQL Server has access to, and the number of worker threads used during a parallel query, one single parallel query can require many worker threads.

If you run into this specific case of high and frequent THREADPOOL waits you will usually see many CXPACKET waits as well (sometimes with high wait times). To show this behavior I have modified the query we used to generate THREADPOOL waits so that it will execute using parallelism. In this case I commented out the MAXDOP query option:

```
SELECT TOP 1 *
FROM Sales.SalesOrderDetail
ORDER BY NEWID()
-- OPTION (MAXDOP 1)
```

For this example I also configured the Max Degree of Parallelism to its default value of 0, and set the Cost Threshold for Parallelism option to 1. This way I am 100% sure the query will be run using parallelism. I left the Max Worker Threads option on a value of 128 as we configured earlier.

If we now repeat the same Ostress test we performed earlier in this chapter by executing the following command, we should see THREADPOOL waits occur again in the sys.dm_os_waiting_tasks DMV, as shown in Figure 5-26.

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -i"C:\select_rnd.sql" -n150 -r10 -q
```

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type
1	0x000000039DFDF7C28	NULL	NULL	323	THREADPOOL
2	0x000000039E4CE9088	NULL	NULL	266	THREADPOOL
3	0x000000039DFDF6108	NULL	NULL	234	THREADPOOL
4	0x000000039EC461848	NULL	NULL	89	THREADPOOL
5	0x000000039E4CE9468	NULL	NULL	81	THREADPOOL
6	0x000000039E80DCCA8	NULL	NULL	69	THREADPOOL
7	0x000000039E4CE9848	NULL	NULL	61	THREADPOOL
8	0x000000039E4CE8108	NULL	NULL	36	THREADPOOL
9	0x000000039E4CE8CA8	NULL	NULL	12	THREADPOOL
10	0x000000039E2F388C8	NULL	NULL	1483	THREADPOOL

Figure 5-26. THREADPOOL waits

But this time, because our test query is executed in parallel, we will also find many CXPACKET waits returned by the sys.dm_os_waiting_tasks DMV, as shown in Figure 5-27.

13	0x000000039E58A9848	146	0	1554	CXPACKET
14	0x000000039E2C2E4E8	83	0	1553	CXPACKET
15	0x000000039E2C2E108	136	0	1553	CXPACKET
16	0x000000039EFE74CA8	NULL	NULL	153	THREADPOOL
17	0x000000039EFE74108	NULL	NULL	128	THREADPOOL
18	0x000000039EC461848	NULL	NULL	88	THREADPOOL
19	0x000000039F5B63468	132	0	1964	CXPACKET
20	0x000000039E8D388C8	133	0	1959	CXPACKET
21	0x000000039EE1D9468	134	0	1985	CXPACKET
22	0x000000039DED0E108	76	0	1985	CXPACKET
23	0x000000039E76F24E8	NULL	NULL	1087	THREADPOOL
24	0x000000039E886E108	NULL	NULL	1022	THREADPOOL
25	0x000000039E2F39468	NULL	NULL	877	THREADPOOL
26	0x000000039DFD81088	NULL	NULL	860	THREADPOOL

Figure 5-27. CXPACKET and THREADPOOL waits

If you see this behavior occurring on your SQL Server instance, it might be worth the effort to check your parallelism configuration. The first section of this chapter discussed CXPACKET waits and how you can lower them. Another hint that might steer you in this direction is that the CPU load during this particular case is usually higher than normal. In the case of my test SQL Server instance, all my CPUs went to 100%.

Lowering THREADPOOL Waits Caused by User Connections

Another common cause of THREADPOOL waits is a sudden increase in the number of users connecting and executing queries against your SQL Server instance. This problem can occur if, for instance, the application that is connecting to your SQL Server instance uses multiple connections. The main problem here is that those connections stay active and keep acquiring worker threads.

To give you an example of this problem we will again use Ostress to connect and execute queries against my test SQL Server instance. In this case we will use a different .sql file, saved as wait.sql, as input for Ostress, with the following query inside it:

```
WAITFOR DELAY '00:05:00'
```

The only thing this query will do is wait for 5 minutes. After those 5 minutes, the query will end and the connection will disconnect.

Let's run Ostress using the wait.sql file:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E  
-dAdventureWorks -i"C:\wait.sql" -n120 -r1 -q
```

We change the number of threads generated by Ostress to 120 and again leave the Max Worker Threads option set to 128 worker threads.

When we query the sys.dm_exec_sessions DMV using the following query, we see that many new user sessions, generated by the Ostress utility, are active, as shown in Figure 5-28.

```
SELECT *  
FROM sys.dm_exec_sessions  
WHERE is_user_process = 1;
```

	session_id	login_time	host_name	program_name
7	57	2019-02-18 11:33:01.527	EVDL-SQL2017-01	OSTRESS
8	58	2019-02-18 11:33:01.527	EVDL-SQL2017-01	OSTRESS
9	59	2019-02-18 11:33:01.527	EVDL-SQL2017-01	OSTRESS
10	60	2019-02-18 11:33:01.530	EVDL-SQL2017-01	OSTRESS
11	61	2019-02-18 11:33:01.537	EVDL-SQL2017-01	OSTRESS
12	62	2019-02-18 11:33:01.537	EVDL-SQL2017-01	OSTRESS
13	63	2019-02-18 11:33:01.533	EVDL-SQL2017-01	OSTRESS
14	64	2019-02-18 11:33:01.537	EVDL-SQL2017-01	OSTRESS
15	65	2019-02-18 11:33:01.537	EVDL-SQL2017-01	OSTRESS

Figure 5-28. Ostress user sessions

If we query the sys.dm_os_waiting_tasks DMV, we see that THREADPOOL waits are occurring, as shown in Figure 5-29.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type
82	0x00000039F3D71468	NULL	NULL	201566	THREADPOOL
83	0x00000039F3D708C8	NULL	NULL	201566	THREADPOOL
84	0x00000039F3D704E8	NULL	NULL	196910	THREADPOOL
85	0x00000039E0FAC4E8	NULL	NULL	177392	THREADPOOL
86	0x00000039E0FAC8C8	NULL	NULL	173108	THREADPOOL
87	0x00000039E0FAD088	NULL	NULL	112424	THREADPOOL
88	0x00000039E0FACCA8	NULL	NULL	23416	THREADPOOL

Figure 5-29. THREADPOOL waits inside the sys.dm_os_waiting_tasks DMV

The big difference between THREADPOOL waits caused by excessive parallelism and an increase in user connections is that the CPU of my test SQL Server instance remains low in the latter case, as shown in Figure 5-30.

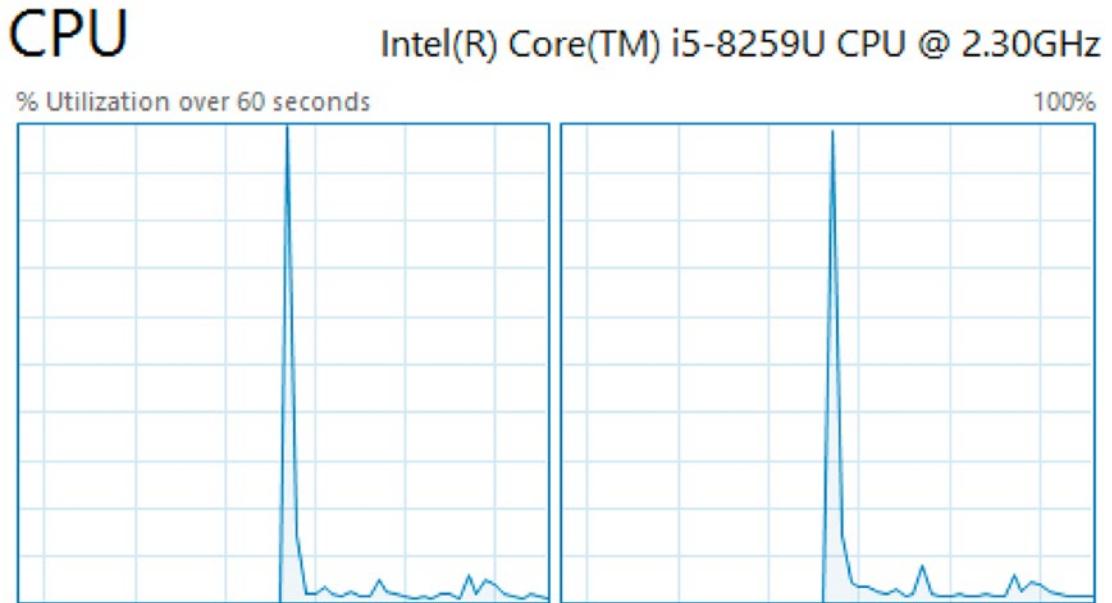


Figure 5-30. CPU usage

The small spike in the CPU usage history graph is caused by starting up the OStress utility. After that, the CPUs remain at a constant low usage percentage.

Resolving THREADPOOL waits caused by an increase in user connections should start at the source. Where do the user connections come from? What are those connections performing? I have seen cases where an application suddenly used hundreds of active user connections after an update, and as the SQL Server instance was not designed to handle that amount of concurrent, active, connections, THREADPOOL waits appeared.

Keep in mind that the user connections should only cause THREADPOOL waits when they are actually running queries. User connections that are connected to the SQL Server instance but are not executing anything should not be a reason for THREADPOOL waits.

Also, having many different user connections active against a database can create many locks on rows or tables. If you notice high lock-related wait times together with THREADPOOL waits, the problem could be the high amount of locking and blocking occurring. In this case you should try to find the queries that are causing the lock waits and see if you can optimize them. We will discuss lock-related wait types, and what you can do about them, in Chapter 7, “Lock-Related Wait Types.”

THREADPOOL Summary

THREADPOOL waits are one of the most alarming wait types to see on your SQL Server instance. They occur because there are not enough free worker threads available to process requests, so tasks that request a worker thread will have to wait until a new worker thread becomes available. Thankfully, THREADPOOL waits are not very common, as they have the potential to completely lock you out of your SQL Server instance. The only way to connect in those cases is by using the Dedicated Administrator Connection (or DAC), which I urge you to enable on all your SQL Server instances.

Excessive use of parallelism and a large increase in active user connections are two of the most common causes for THREADPOOL waits. The former has a direct relation to the CXPACKET wait type we discussed earlier, so methods to resolve the CXPACKET wait type can also help to resolve THREADPOOL waits. The latter requires a deeper investigation into why the number of active user connections suddenly increased. Maybe they are the result of a bug in the application connection to the SQL Server instance. We also briefly touched on locking and blocking behavior as a possible cause for THREADPOOL waits. We will take a deeper look at how we can resolve lock-related waits in Chapter 7, “Lock-Related Wait Types.”

CHAPTER 6

IO-Related Wait Types

In this chapter we will take a look at IO-related wait types in the broadest sense of the term. I selected wait types that are related to the storage, memory, or network components of your system. One could argue that the majority of wait types will fit this category, and that is probably right, but to prevent this chapter from covering 90% of all the wait types in this book, I had to choose carefully. I consider the wait types in this chapter to have a direct relation to storage, memory, or network but to not relate directly to a functionality or concept in SQL Server. For instance, the PAGEIOLATCH_xx wait types are frequently related to storage, but they are not included in this chapter. The reason for this is because they are also a latch wait type, and I believe latch wait types deserve a separate chapter because of their function in SQL Server.

The performance of the IO-related components is incredibly important for SQL Server. Practically every part of SQL Server interacts with these components in one way or another, whether it is a data page that needs to be read from disk into memory or the results from a query that need to be transported across the network to your end users. If one of these components can't handle the workload you are generating on your SQL Server instance, or isn't configured properly, your performance will decline.

The wait types in this chapter can help you track down which of your IO components is slowing you down so you can take appropriate action to prevent or resolve performance-related incidents.

ASYNC_IO_COMPLETION

The ASYNC_IO_COMPLETION wait type is a pretty common wait type that occurs every time SQL Server performs a file-related action on the storage subsystem and has to wait for it to complete. You will frequently see this wait type when you are performing actions that interact with the storage subsystem, like a backup. Just like with most wait types, if you are seeing this wait type occur, it doesn't necessarily mean there is a problem with

your storage subsystem. It will only become a problem if the wait time is longer than you expect it to be compared to your baseline values, which we discussed in Chapter 4, “Building a Solid Baseline.”

What Is the ASYNC_IO_COMPLETION Wait Type?

If we look up the ASYNC_IO_COMPLETION wait type in Books Online (BOL), we will see the following definition: “Occurs when a task is waiting for I/Os to finish.” This is a rather short and vague definition. Let’s add a little more detail. ASYNC_IO_COMPLETION waits occur when a task is waiting for a storage-related action to finish. The task is initiated and monitored by SQL Server. Figure 6-1 shows this as a visual representation of the wait type.

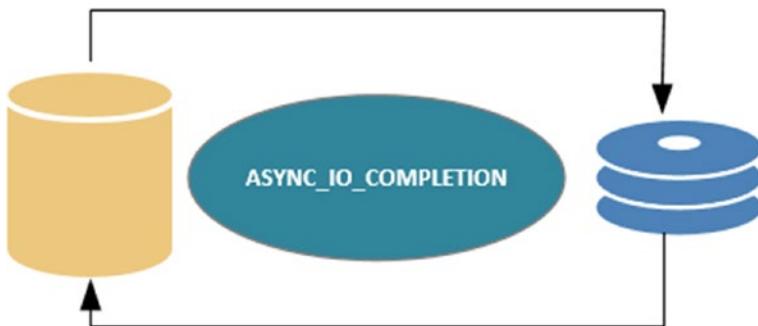


Figure 6-1. ASYNC_IO_COMPLETION wait occurring

As long as the storage-related action is running, the ASYNC_IO_COMPLETION wait time is being logged. As you can imagine, the faster your storage subsystem, the lower your ASYNC_IO_COMPLETION wait times will be.

As I said earlier, usually ASYNC_IO_COMPLETION waits are no cause for concern. They will happen normally during many SQL Server operations that need to access the storage subsystem, like backups or the creation of a new database. It can become a cause for concern if the wait times are higher than you expected when compared to your baseline measurements.

ASYNC_IO_COMPLETION Example

Let's go through an example that generates ASYNC_IO_COMPLETION waits. We won't need any extra utilities for this; just running a database backup will trigger ASYNC_IO_COMPLETION waits.

In this case I will perform a backup of the AdventureWorks database on my test server.

To perform this action I will use the query in Listing 6-1. This query will reset the sys.dm_os_wait_stats DMV, perform the database backup, and then query the sys.dm_os_wait_stats DMV for the ASYNC_IO_COMPLETION waits.

Listing 6-1. Generate ASYN_IO_COMPLETION waits

```
USE [master]
GO

DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

BACKUP DATABASE [AdventureWorks]
TO DISK = N'F:\Backup\aw_backup.bak'
WITH
NAME = N'AdventureWorks-Full Database Backup',
STATS = 2;

GO

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'ASYNC_IO_COMPLETION';
```

The backup operation took 1 second on my test SQL Server instance. Figure 6-2 shows the results of the query in Listing 6-1.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_IO_COMPLETION	2	1126	1126	0

Figure 6-2. ASYNC_IO_COMPLETION wait time

As you can see, for almost the entire duration of the database backup ASYNC_IO_COMPLETION waits were logged.

Lowering ASYNC_IO_COMPLETION Waits

One common cause of high ASYNC_IO_COMPLETION wait times is a database backup, as you just saw in the example. If you want to find out if your ASYNC_IO_COMPLETION waits are occurring because a backup is being performed, try to look for backup-related waits occurring at the same time.

If we were to slightly modify the last sys.dm_os_waiting_tasks query in Listing 6-1, we would see backup-related wait types being returned by the DMV:

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type IN
(
    'ASYNC_IO_COMPLETION',
    'BACKUPIO',
    'BACKUPBUFFER'
);
```

Figure 6-3 shows the result of the query in Listing 6-1, but with this modification.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_IO_COMPLETION	2	1126	1126	0
2	BACKUPBUFFER	42	167	41	6
3	BACKUPIO	184	779	42	4

Figure 6-3. ASYNC_IO_COMPLETION waits together with backup-related waits

If you see both occurring at the same time, chances are that a database backup is causing your ASYNC_IO_COMPLETION waits.

Another possible method of lowering ASYNC_IO_COMPLETION waits is by configuring instant file initialization. Instant file initialization was introduced in Windows 2003 and speeds up the process of allocating space on a disk tremendously by removing the need to zero-out files (writing zeros inside files before they can get used). This does not affect the speed of your backup, but will give increased performance when creating a database, adding files to a database, or restoring a database. Instant file initialization is not enabled by default, unless you are running your SQL Server service under an account that has

local administrator privileges. During the setup of SQL Server 2016, Microsoft added an additional checkbox to enable instant file initialization during SQL Server setup, as shown in Figure 6-4.

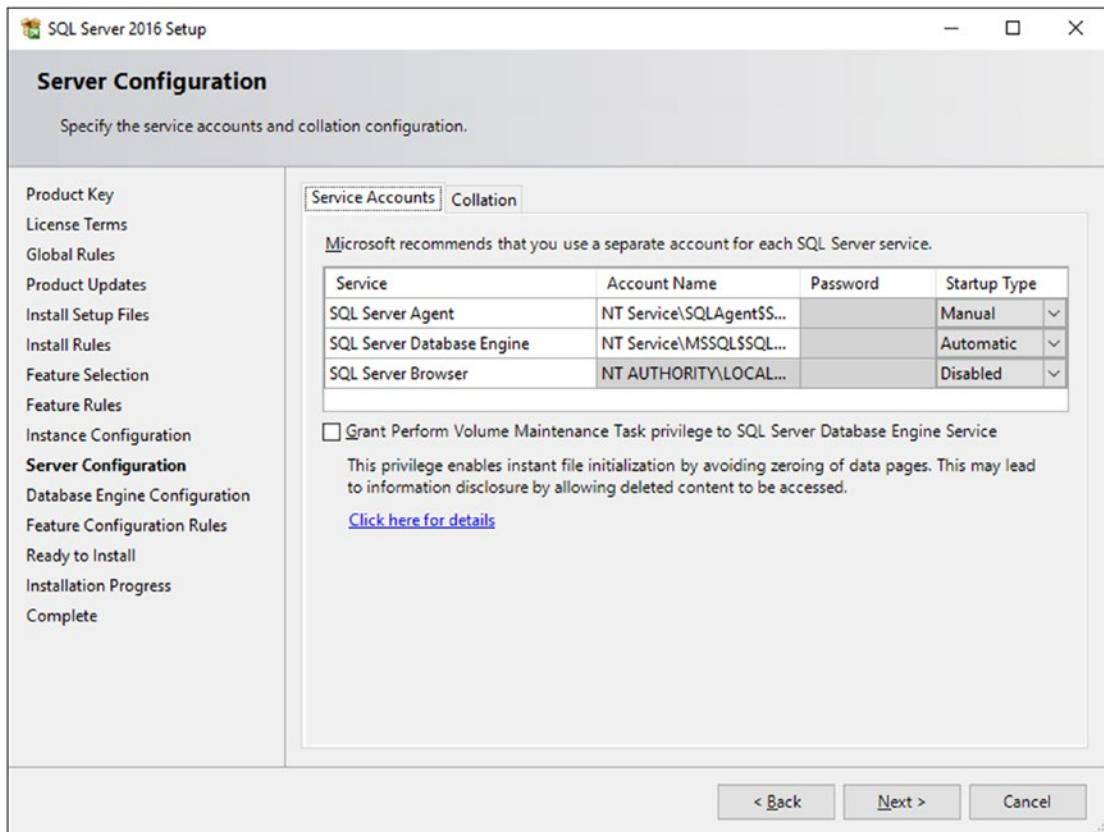


Figure 6-4. Grant Perform Volume Maintenance Task privilege to the SQL Server Database Engine Service checkbox in SQL Server 2016 setup

If you didn't enable the Grant Perform Volume Maintenance Task privilege to the SQL Server Database Engine Service checkbox during the installation of SQL Server 2016 or higher, or installed a lower version of SQL Server, you will have to configure instant file initialization manually after installation. The way to configure instant file initialization is through a local security policy on the machine SQL Server is running on by adding the account your SQL Server service is running under.

You can find this policy by opening the Local Security Policy MMC under Administrative Tools in the Configuration Panel. Open up the Local Policies ➤ User Rights Assignment folder and scroll down to the “Perform volume maintenance tasks” policy, as shown in Figure 6-5.

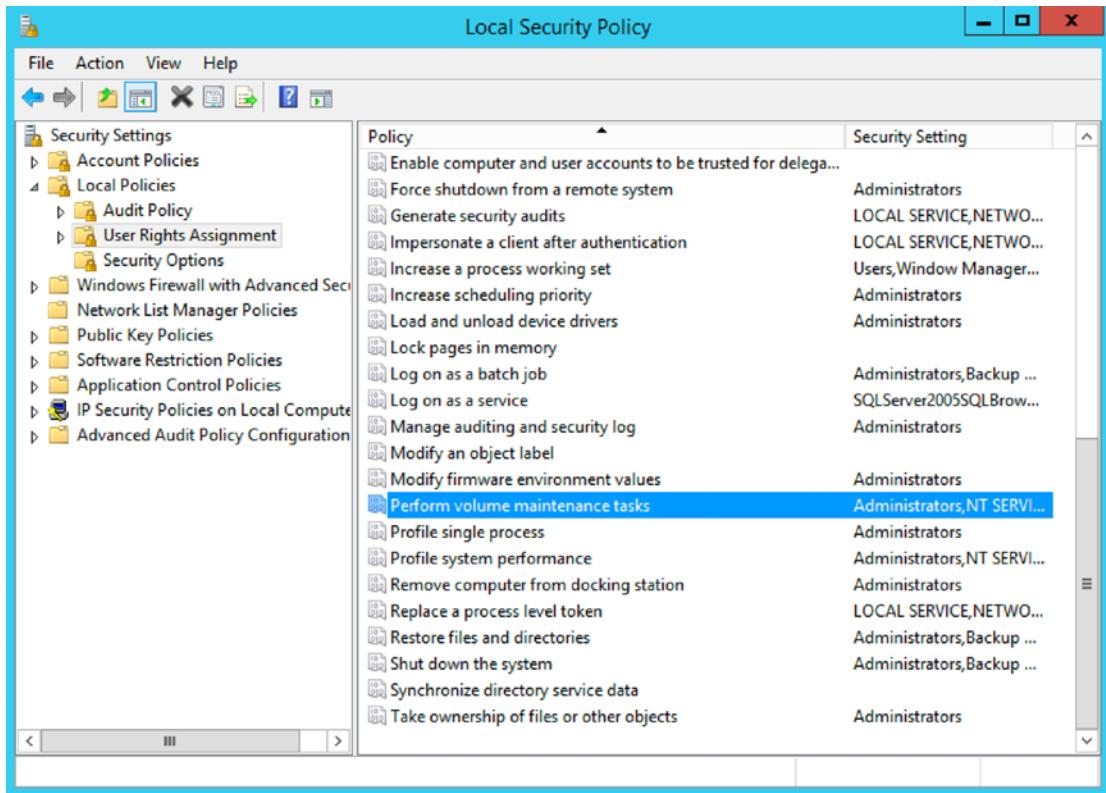


Figure 6-5. Perform volume maintenance tasks local policy

Double-click the policy to open it and add the account your SQL Server service is running under. The last step is restarting your SQL Server service. After the restart, SQL Server can make use of instant file initialization.

To show you the impact of instant file initialization, I used the query in Listing 6-2. This query clears the sys.dm_os_wait_stats DMV, then creates a new database with a 500 MB data file and a 100 MB log file. It then queries the sys.dm_os_wait_stats DMV for the ASYNC_IO_COMPLETION wait type.

Listing 6-2. Measure the impact of instant file initialization on ASYNC_IO_COMPLETION waits

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

CREATE DATABASE [IO_test]
ON PRIMARY
(
    NAME = N'IO_test', FILENAME = N'E:\Data\IO_test.mdf' , SIZE = 512000KB ,
    FILEGROWTH = 10%
)
LOG ON
(
    NAME = N'IO_test_log', FILENAME = N'E:\Log\IO_test_log.ldf' ,
    SIZE = 102400KB , FILEGROWTH = 10%
);
GO

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'ASYNC_IO_COMPLETION';
```

Figure 6-6 shows the wait statistics information both before and after configuring instant file initialization.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_IO_COMPLETION	1	10591	10591	0

Before

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_IO_COMPLETION	1	1878	1878	0

After

Figure 6-6. Impact of instant file initialization on ASYNC_IO_COMPLETION waits

Even for this relatively small database the gain of using instant file initialization is pretty big, as you can see in the difference in wait times. Before enabling instant file initialization, the query in Listing 6-2 took 11 seconds to complete; after the change it went down to 2 seconds.

If you configured instant file initialization and checked that no backups are being performed at the same time that you are seeing high ASYNC_IO_COMPLETION waits, the problem might be your storage subsystem. A good method of analyzing potential storage problems is by using Perfmon to monitor the Avg. Disk/sec Read and Avg. Disk/sec Write counters on the disks on which your database resides, as shown in Figure 6-7. These counters show you the read and write latency to your disks in seconds (this means a value of 0.005 means 5 milliseconds). SQL Server performs optimally with a maximum latency of 5 milliseconds. Above 20 milliseconds, latency will cause noticeable performance degradation. The higher the latency value, the higher the wait time of storage-related wait types will be.

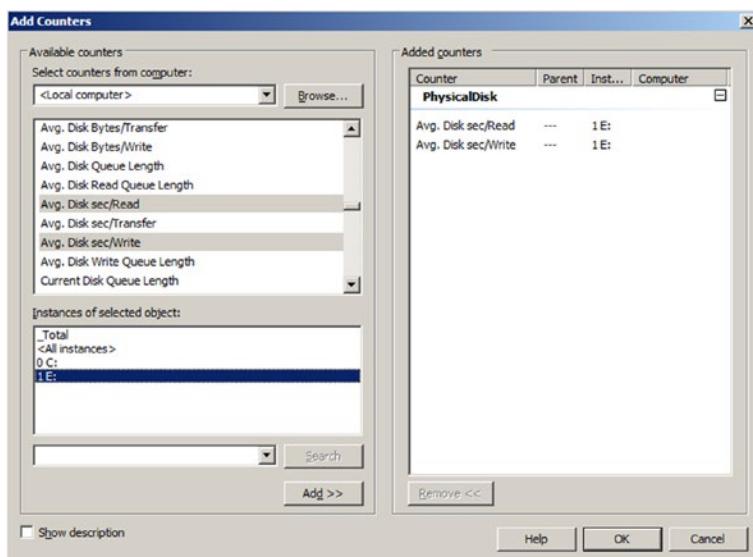


Figure 6-7. Avg. Disk sec/Read and Avg. Disk sec/Write Perfmon counters

Be careful about jumping to conclusions regarding your storage performance. Always talk to your storage administrator (if you have one) and show him/her your measurements before you decide the storage subsystem is the bottleneck. Storage is the domain of the storage administrator, and he/she can help you analyze and resolve performance problems.

ASYNC_IO_COMPLETION Summary

The ASYNC_IO_COMPLETION wait type occurs when you perform actions related to the storage subsystem from inside your SQL Server instance, most notably database backups and the creation of new databases. While ASYNC_IO_COMPLETION waits are completely normal, they can indicate storage-related problems if wait times are higher than normal. Before you run to your storage administrator, make sure there is actually a performance problem. One possible way to do this is by checking your storage latency, as high latency values will impact ASYNC_IO_COMPLETION wait times as well. Also check whether the higher ASYNC_IO_COMPLETION wait times are directly related to database backups being performed. One great method to lower ASYNC_IO_COMPLETION wait times is by enabling instant file initialization by adding your SQL Server service account to the Perfmon volume maintenance tasks local policy.

ASYNC_NETWORK_IO

Just like the ASYNC_IO_COMPLETION wait type, the ASYNC_NETWORK_IO wait type is related to throughput. But instead of storage subsystem throughput, the ASYNC_NETWORK_IO wait type is related to the throughput of your network connection between your SQL Server instance and your clients. Again, seeing wait times for this specific wait type does not necessarily mean there is a network-related issue, since ASYNC_NETWORK_IO waits always occur, even if you query your SQL Server instance on the SQL Server itself.

What Is the ASYNC_NETWORK_IO Wait Type?

ASYNC_NETWORK_IO waits usually occur when client applications cannot process the query results fast enough, or when you have a network-related performance problem. The former will in most cases be the most likely, since many applications process SQL Server results on a row-by-row basis, or simply cannot handle the amount of data. This forces the SQL Server to wait on sending query results across the network. While SQL Server is waiting to send the requested data, the ASYNC_NETWORK_IO wait type is logged. Another situation in which ASYNC_NETWORK_IO waits can occur is when you are using a linked server to query remote databases. Figure 6-8 shows a graphical representation of this.

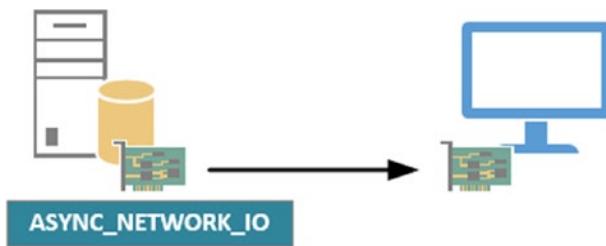


Figure 6-8. ASYNC_NETWORK_IO

ASYNC_NETWORK_IO Example

Showing an example of the ASYNC_NETWORK_IO wait type doesn't require a complicated test environment. Listing 6-3 shows a query that will generate ASYNC_NETWORK_IO waits when run against my test SQL Server instance from another computer using SQL Server Management Studio, which should be enough. The query is going to clear the sys.dm_os_wait_stats DMV, then perform the actual query against the AdventureWorks database. The last statement will show us the wait times of the ASYNC_NETWORK_IO wait type.

Listing 6-3. Generate ASYNC_NETWORK_IO waits

```
DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);

SELECT *
FROM Person.Person;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'ASYNC_NETWORK_IO';
```

Figure 6-9 shows the wait times for the ASYNC_NETWORK_IO wait type.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_NETWORK_IO	165	516	58	50

Figure 6-9. ASYNC_NETWORK_IO wait times

In this example, the results of the query against the AdventureWorks database couldn't be processed by the SQL Server Management Studio application as fast as the SQL Server instance supplied the results, and ASYNC_NETWORK_IO waits occurred.

Lowering ASYNC_NETWORK_IO Waits

One of the “easiest” ways to lower ASYNC_NETWORK_IO waits is to identify queries that will return a large result set back to the application. For instance, if we modify the query to return only the first 100 rows, the SQL Server Management Studio might be able to keep up with the information returned to it:

```
DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);

SELECT TOP 100 *
FROM Person.Person;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'ASYNC_NETWORK_IO';
```

The resulting wait times after this modification can be seen in Figure 6-10.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_NETWORK_IO	0	0	0	0

Figure 6-10. ASYNC_NETWORK_IO wait times after modifying the query

As you can see, we didn't run into any ASYNC_NETWORK_IO waits this time. The SQL Server Management Studio was able to keep up with the results returned, so the SQL Server instance we queried didn't have to delay sending the results back to the client.

Another way to limit results returned could be by filtering out information using WHERE clauses that isn't used by the application in the first place. Smaller results will result in lower ASYNC_NETWORK_IO wait times.

If you believe that ASYNC_NETWORK_IO waits are not caused by large results being returned to an application, or by the speed at which an application can process the results, there is also a possibility that your network configuration is slowing you down. In this case you should first check your network utilization. Sadly, there isn't a counter

CHAPTER 6 IO-RELATED WAIT TYPES

in Perfmon that directly shows the network utilization without having you perform some math to calculate it. Instead, you can use the Networking tab of the Task Manager to view your network-card utilization, as shown in Figure 6-11.

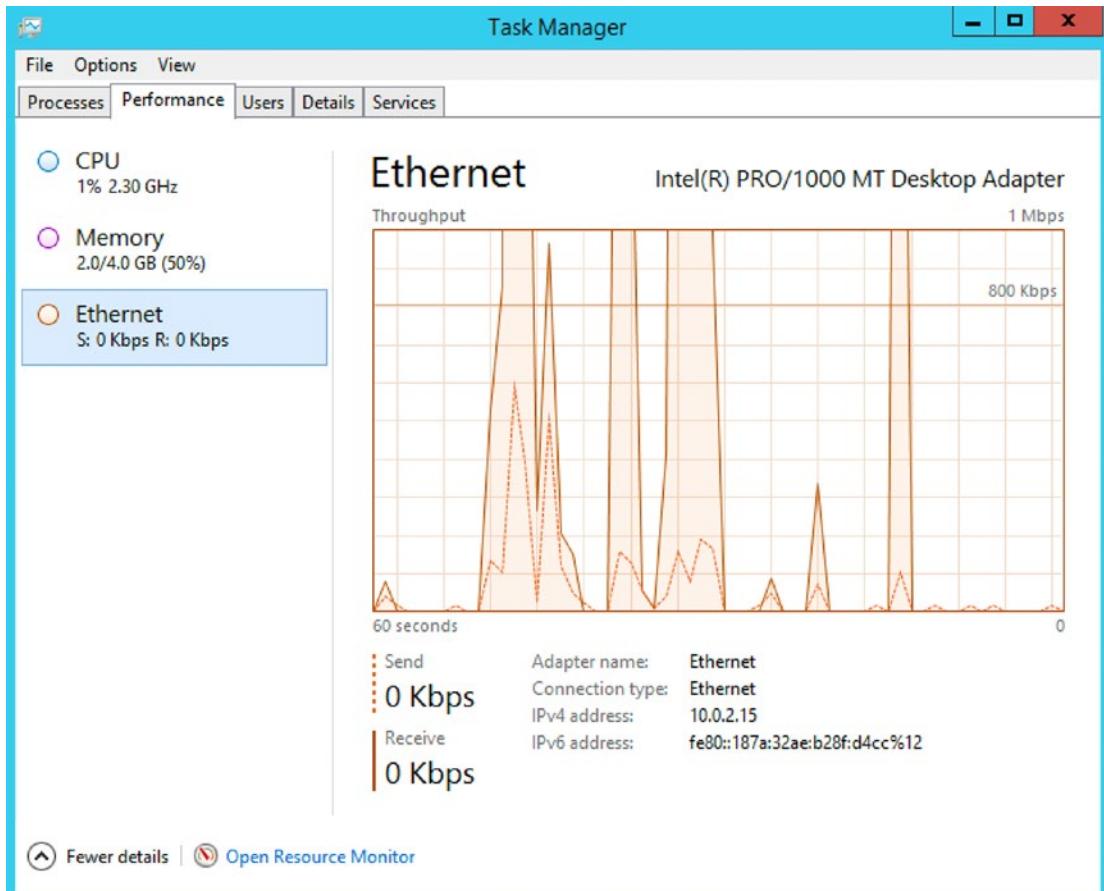


Figure 6-11. Task Manager network utilization

If you notice that the network utilization is high while you are experiencing higher than normal `ASYNC_NETWORK_IO` wait times, it could be possible that the network is slowing you down. In that case it might be a good idea to talk to your network administrator. Usually a network configuration consists of many parts, like switches, routers, firewalls, network cables, drivers, firmware, potential virtualization of the operating system, and so on. All of these parts can slow down your network throughput and can be a potential cause of `ASYNC_NETWORK_IO` waits.

ASYNC_NETWORK_IO Summary

The ASYNC_NETWORK_IO wait type occurs whenever an application requests query results from a SQL Server instance over the network and cannot process the returned results fast enough. Seeing ASYNC_NETWORK_IO waits occur is completely normal, but higher than normal wait times can be caused by changes in the returned query results or network-related problems. Lowering ASYNC_NETWORK_IO wait times that are application related can be achieved by decreasing the number of rows and/or columns returned to the application.

CMEMTHREAD

Waits of the CMEMTHREAD wait type are memory related and indicate a pressure on certain SQL Server-related memory objects. These memory objects allocate memory for the various parts of SQL Server like the buffer cache and the procedure cache. Whenever CMEMTHREAD waits occur, it means that multiple threads are trying to access the same memory object at the same time.

What Is the CMEMTHREAD Wait Type?

To explain how CMEMTHREAD wait type generation works, we have to dig a little deeper inside some programming terminology, specifically the terms *mutual exclusions*, *critical sections*, and *thread safety*. These three concepts play a direct role in the CMEMTHREAD wait type generation.

A critical section consists of a piece of code that accesses a shared resource that can only be accessed by one thread at a time. In our case the shared resource would be a SQL Server memory object. The SQL Server memory objects can only be accessed one thread at a time so as to ensure no corruption to the memory object can occur. Because there are many threads that want access to memory objects, we have to use a method to ensure only one thread gets access at a time. This method is called mutual exclusion. SQL Server uses a Mutex object to make sure concurrent threads are not in their critical sections at the same time when accessing the memory object. A Mutex does this by serializing the thread access to the memory object. Only a single thread can be the owner of a Mutex object, and while a thread has ownership, it can access the shared resource. When the thread is done, the Mutex object will move to the next thread in line. By using these objects we have created a thread-safe code, where multiple threads do not have concurrent access to memory objects. Figure 6-12 shows this situation.

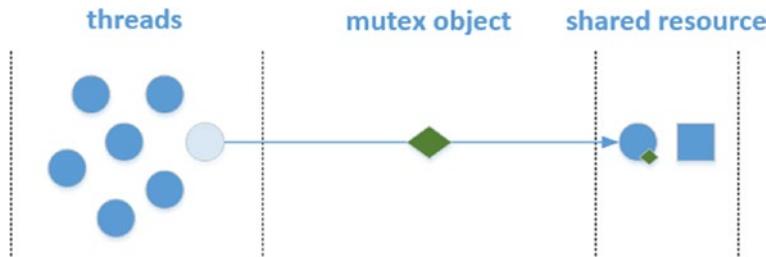


Figure 6-12. Thread waiting for a Mutex object to access a shared resource

A simplified example of this behavior is when you and a large group of other people are waiting for a single ticket dispenser to buy a ticket to see your favorite rock band. In this case the ticket dispenser is the shared resource, and only a single person can access the ticket dispenser at a time. When we reach the ticket dispenser, we can get a ticket, and the people behind us have to wait till it's their time. After we have bought a ticket, the next person in line gets access to the ticket dispenser.

We can also view this behavior in SQL Server, but to do this we have to make use of a debugger (like WinDbg). Figure 6-13 shows how a thread in SQL server waits for a Mutex since access to the memory object was granted to another thread. To capture this image I used an Extended Event session that created a SQL Server mini-dump when a CMEMTHREAD wait occurred. I then used WinDbg to open the mini-dump and returned the stack.

```
0:170> ln 0x000007FEF9F86059;ln 0x000007FEF9F4219C;ln 0x000007FEF9F41F03;ln 0x000007FEF9F437AB;
(000007fe`f9f41400) sqldk!XeSosPkg::wait_info::Publish+0x138  | (000007fe`fa020930) sqldk
(000007fe`f9f41f70) sqldk!SOS_Scheduler::UpdateWaitTimeStats+0x2bc  | (000007fe`f9f42250)
(000007fe`f9f41e90) sqldk!SOS_Task::PostWait+0x9e  | (000007fe`f9f4db60) sqldk!SOS_Task:::
(000007fe`f9f437e0) sqldk!EventInternal<SuspendQueueSlock>::Wait+0x2ca  | (000007fe`f9f43a6
(000007fe`f9f461a0) sqldk!SOS_UnfairMutexPair::LongWait+0x191  | (000007fe`f9f6e400) sqldk
(000007fe`f9f455d0) sqldk!SOS_UnfairMutexPair::AcquirePair+0x46  | (000007fe`f9f45640) sqldk
(000007fe`f9f45970) sqldk!CMemThread<CMemObj>::Alloc+0xb6  | (000007fe`f9fdee40) sqldk!CM
```

Figure 6-13. Example of a CMEMTHREAD wait occurring in a mini-dump

The important line here is the `SOS_UnfairMutexPair::LongWait`, which generates the CMEMTHREAD wait because the thread we are monitoring here has to wait for another thread that currently has access to the memory object. The line after that, `SOS_UnfairMutexPair::AcquirePair`, means the thread received the Mutex, followed by access to the memory object represented by `CMemThread<CMemObj>::Alloc`.

Lowering CMEMTHREAD Waits

Since there are many different memory objects present in SQL Server that could potentially generate CMEMTHREAD waits, there are many possible solutions to lowering CMEMTHREAD wait times depending on the memory object that is being accessed.

One of the more common situations where CMEMTHREAD waits can occur is when large amounts of short, concurrent, ad hoc queries are being executed. Every time an ad hoc query is executed that could not be parameterized, the Query Optimizer will generate a new execution plan for the query. All these new execution plans need to be entered into the procedure cache, and a memory object for allocating cache descriptors is accessed. Since the memory object is thread-safe, CMEMTHREAD waits can occur if the rate of insertion is high enough. A good place to start looking if you suspect CMEMTHREAD waits are going to occur because of ad hoc queries is the procedure cache. The query in Listing 6-4 will give you information about the number of execution plans in the procedure cache.

Listing 6-4. Query procedure cache

```
SELECT
    objtype,
    COUNT_BIG (*) AS 'Total Plans',
    SUM(CAST(size_in_bytes AS DECIMAL(12,2))/1024/1024) AS 'Size (MB)'
FROM sys.dm_exec_cached_plans
GROUP BY objtype;
```

The results of this query should look like Figure 6-14, though the numbers will be different on your system.

	objtype	Total Plans	Size (MB)
1	Prepared	36	12.585937
2	View	316	43.070312
3	Adhoc	350	45.132812
4	Proc	6	0.500000

Figure 6-14. Results of querying procedure cache

We should focus on the number of ad hoc execution plans. If you see this number growing rapidly and experience CMEMTHREAD waits, it might be worth the effort to analyze some of those ad hoc queries. If possible, try to optimize the queries so they generate a reusable plan. If your application uses many dynamic queries then try to use the sp_executesql system-stored procedure instead of the EXECUTE (EXEC) command. Using the EXEC command will most likely result in a plan that will only be used once.

Microsoft has released various fixes (most notably the partitioning of certain memory objects across CPUs) for this problem in SQL Server 2005 SP2, making it less common these days. Even if you are using newer SQL Server editions than SQL Server 2005, it might be a good idea to upgrade to the latest available Service Pack since there have been various memory-related bug fixes in every SQL Server edition.

CMEMTHREAD Summary

The CMEMTHREAD wait type is a memory-related wait type. CMEMTHREAD waits occur when multiple threads try to access memory objects that can only be accessed by one thread at a time. The time other threads spend waiting for their turn to access the memory object is recorded as CMEMTHREAD wait time. One of the more common cases where CMEMTHREAD waits can occur is when your system uses a high amount of ad hoc queries. Every time a new execution plan is generated, SQL Server will access a memory object; if many execution plans are generated this can lead to a queue of threads that want access to the memory object, resulting in CMEMTHREAD waits.

IO_COMPLETION

Just like the ASYNC_IO_COMPLETION wait type, IO_COMPLETION waits occur when SQL Server is waiting for storage-related actions to complete. And just like the ASYNC_IO_COMPLETION wait type, seeing high wait times of the IO_COMPLETION wait type doesn't necessarily mean there is something wrong with your storage system. IO_COMPLETION waits occur normally while your SQL Server instance is running and should only be a concern if wait times are a lot higher than normal.

What Is the IO_COMPLETION Wait Type?

While the ASYNC_IO_COMPLETION wait type is recorded when database-related actions are performed, like a database backup, IO_COMPLETION waits occur when non-data pages are involved, like the restore of a transaction log backup or when bitmap allocation pages, like the GAM page, are accessed. IO_COMPLETION waits can also occur when queries are being executed that perform read or write operations to the storage subsystem, like a Merge Join operator.

IO_COMPLETION Example

Let's generate some IO_COMPLETION waits by restoring a transaction log backup. For this example we will make use of the AdventureWorks database again. The query in Listing 6-5 will perform a full backup of the AdventureWorks database, make some changes, and perform a transaction log backup. When that is complete we will restore the full backup again, clear the sys.dm_os_wait_stats DMV, restore the transaction log backup, and check for IO_COMPLETION waits.

Listing 6-5. Generate IO_Completion waits

```
-- Make sure AdventureWorks is in Full recovery model
ALTER DATABASE AdventureWorks SET RECOVERY FULL
GO

-- Perform full backup first
-- Otherwise FULL recovery model will not be affected
BACKUP DATABASE [AdventureWorks]
TO DISK = N'F:\Backup\AW_Full.bak'
GO

-- Make some changes to AW database
USE AdventureWorks
GO

UPDATE Person.Address
SET City = 'Portland'
WHERE City = 'Bothell'
```

CHAPTER 6 IO-RELATED WAIT TYPES

```
-- Backup Transaction Log
BACKUP LOG [AdventureWorks]
TO DISK = N'F:\Backup\AW_Log.trn'
GO

-- Restore the previous full backup with NORECOVERY
USE [master]
GO

RESTORE DATABASE [AdventureWorks]
FROM DISK = N'F:\Backup\AW_Full.bak'
WITH NORECOVERY, REPLACE
GO

-- Clear sys.dm_os_wait_stats
dbcc sqlperf ('sys.dm_os_wait_stats', CLEAR)

-- Restore last Transaction Log backup
RESTORE LOG [AdventureWorks] FROM DISK = N'F:\Backup\AW_Log.trn'
GO

-- Check IO_COMPLETION waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'IO_COMPLETION'
```

The results of this query against the `sys.dm_os_wait_stats` DMV on my test system can be seen in Figure 6-15.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	IO_COMPLETION	132	8	1	0

Figure 6-15. *IO_COMPLETION* waits

We only modified a few records using the query in Listing 6-5, so the total wait time is pretty low since the restore of the transaction log backup occurred fast.

`IO_COMPLETION` wait times also occur when you are starting up your databases after, for instance, a restart of the SQL Server service. This means you should expect `IO_COMPLETION` waits after a restart or a failover; these are completely normal. Also when `AUTO_CLOSE` is enabled on a database (default in Express versions) and a database is starting, you should experience `IO_COMPLETION` waits.

Lowering `IO_COMPLETION` Waits

Most of the time `IO_COMPLETION` waits shouldn't be a cause for concern. When they are a lot higher than the wait times in your baseline, you should analyze the storage subsystem performance like I described in the “Lowering `ASYNC_IO_COMPLETION`” section. While certain query operations can also cause `IO_COMPLETION` waits, these are frequently not the cause for higher-than-normal wait times.

`IO_COMPLETION` Summary

Just like the `ASYNC_IO_COMPLETION` wait type, `IO_COMPLETION` waits occur when accessing your storage subsystem. `IO_COMPLETION` waits occur when SQL Server is waiting on non-data page operations to complete, like a transaction log restore operation or the reading of bitmap pages, like the GAM page. Seeing waits of the `IO_COMPLETION` type occur is completely normal and these frequently do not require deeper analysis unless the wait times are a lot higher than the values in your baseline. In those cases focus on the performance (and especially latency) of your storage subsystem first.

LOGBUFFER and WRITELOG

I have combined both `LOGBUFFER` and `WRITELOG` in this section. This is because both wait types have a close relation to each other. Both of them are related to the transaction log and the storage subsystem.

What Are the LOGBUFFER and WRITELOG Wait Types?

To understand what the LOGBUFFER and WRITELOG wait types represent, we need to have some understanding of how SQL Server writes to the transaction log. In short, the following events happen whenever we change or add data inside a database:

1. Data page where the data resides is modified in the buffer cache; if the page wasn't already in the buffer cache, it will get read into the buffer cache first.
2. The data page will be marked as "dirty" inside the buffer cache.
3. The log records that represent the modification get saved in the log buffer.
4. A log flush occurs (this can be for multiple reasons, which we will discuss later), writing the log records from the log buffer to the transaction log.
5. The dirty data page gets written to the data file.

To show this behavior I have included Figure 6-16.

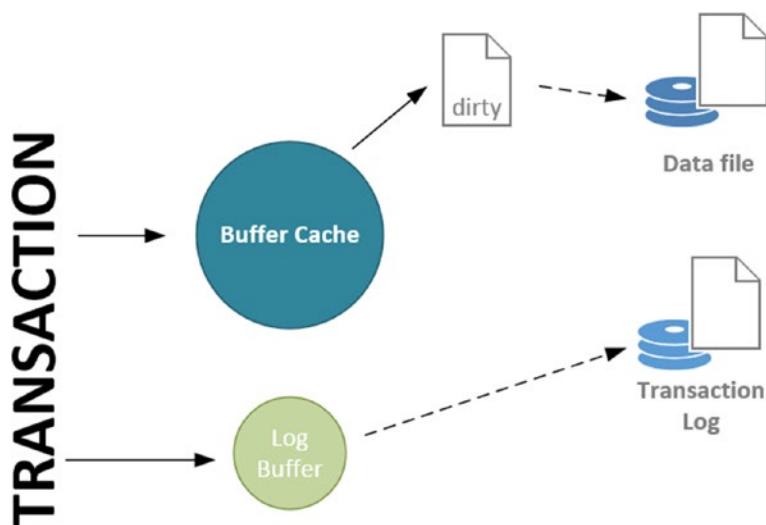


Figure 6-16. How a transaction moves

The action representing the movement of a dirty data page and the action of writing the log records to the transaction log are shown as dashed lines. I did this on purpose to illustrate that both of these actions do not necessarily happen directly.

As you probably know, dirty data pages are updated inside the buffer cache first, and are only written to the data file when a checkpoint operation occurs. This means that a dirty page can stay inside the memory even after your transaction was committed.

This is not true for log records inside the log buffer. As soon as your transaction commits, and that transaction has an active log record in the log buffer, all the log records inside that log buffer are written (or flushed) to the transaction log on disk. But this doesn't only occur when the transaction is committed. The log buffer has a fixed size of 60 KB, and as soon as the log buffer is full, it will flush all the records inside it to the transaction log.

Let's add both wait types we are discussing in this section to the story. The `WRITELOG` wait type occurs whenever SQL Server is flushing the contents of the log buffer to the transaction log on disk. The `LOGBUFFER` wait type occurs when inserting log records in the log buffer, when at the time of insertion SQL Server has to wait for free space inside the log buffer. I have added both the wait types at the parts where they can get generated in Figure 6-17.

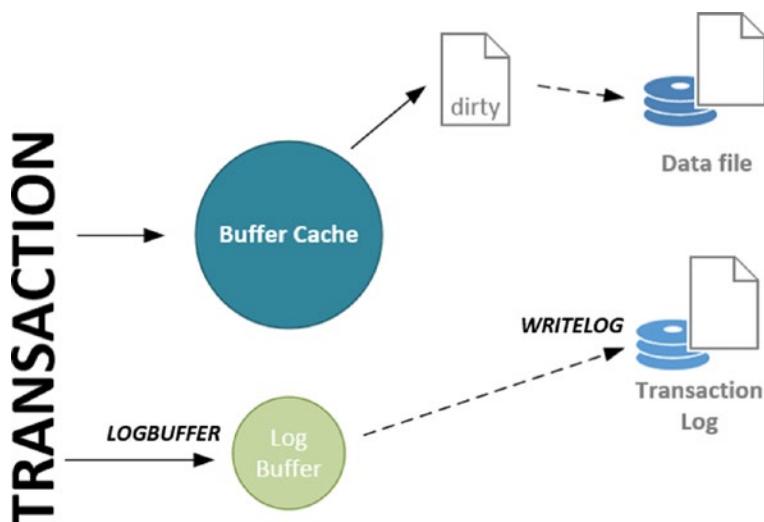


Figure 6-17. Transaction movement and the `LOGBUFFER` and `WRITELOG` wait types

You can now probably see how both wait types are related. Whenever a long WRITELOG wait occurs, chances are you will also see LOGBUFFER waits if the process that is writing the log records to the transaction log on disk cannot process them as fast as the log records enter the log buffer.

This situation frequently occurs on systems with a lot of concurrent data modifications. This results in a high volume of transactions that need to be written to disk. Another common cause is the performance of the storage subsystem where the transaction log file resides. If the storage subsystem has suboptimal performance, your WRITELOG wait times will increase, with the possibility existing that LOGBUFFER waits can occur if the volume of transactions is high enough.

The performance of your transaction log is critical for the performance of your entire database. Slow transaction log performance will have an impact on every change you perform inside your database, as every modification has to be written to the transaction log before it can get committed.

LOGBUFFER and WRITELOG Example

To give you an example of LOGBUFFER and WRITELOG waits occurring, I am creating a new database using the script in Listing 6-6.

Listing 6-6. Create trans_demo database

```
USE master
GO
-- Create demo database
CREATE DATABASE [trans_demo]
ON PRIMARY
(
    NAME = N'trans_demo', FILENAME = N'D:\Data\trans_demo.mdf' , SIZE =
    153600KB , FILEGROWTH = 10%
)
LOG ON
(
    NAME = N'trans_demo_log', FILENAME = N'D:\Log\trans_demo.ldf' , SIZE =
    51200KB , FILEGROWTH = 10%
)
```

```

GO

-- Make sure recovery model is set to full
ALTER DATABASE [trans_demo] SET RECOVERY FULL
GO

-- Perform full backup first
-- Otherwise FULL recovery model will not be affected
BACKUP DATABASE [trans_demo]
TO DISK = N'F:\Backup\trans_demo_Full.bak'
GO

-- Create a simple test table
USE trans_demo
GO

CREATE TABLE transactions
(
    t_guid VARCHAR(50)
)
GO

```

Now that we have created a brand new database, I am going to use the Ostress utility to generate load against the trans_demo database. I am going to execute the query in Listing 6-7, which I saved to the logbuffer_impl.sql file, with 200 concurrent connections using the following command "C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E -dtrans_demo -i"C:\logbuffer_impl.sql" -n200 -r1 -q.

Listing 6-7. Insert rows inside the trans_demo database

```

DECLARE @i INT
SET @i = 1

WHILE @i < 10000

BEGIN

```

```

INSERT INTO transactions
    (t_guid)
VALUES
    (newid())

SET @i = @i + 1

END

```

Before I started the Ostress utility, I cleared the sys.dm_os_wait_stats DMV.

After about 1 minute on my test SQL Server, the Ostress utility finished executing the workload. If I query the sys.dm_os_wait_stats DMV and look for the LOGBUFFER and WRITELOG wait types, I get the results shown in Figure 6-18.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WRITELOG	2000883	6196758	1127	4322304
2	LOGBUFFER	71	1480	34	213

Figure 6-18. WRITELOG and LOGBUFFER waits

Lowering LOGBUFFER and WRITELOG Waits

There are generally two approaches you can take to lowering LOGBUFFER and WRITELOG waits, keeping in mind though that WRITELOG waits do also occur normally and its wait time should only be a cause for concern if it is a lot higher than normal.

The first approach is to take a good look at how your transactions are being executed. In the preceding example, we implicitly committed every INSERT statement. This means that as soon as the log record for the INSERT statement entered the log buffer it needed to be flushed again. If we would explicitly commit the whole WHILE loop we would have larger writes to flush to the transaction log, resulting in better performance. This is because writing small blocks frequently is generally slower than writing large blocks at a larger interval. Cursors can also have the same effect as the example we used, so use them as little as possible.

The other approach is based on the storage subsystem. If SQL Server cannot write the log records fast enough, you can encounter LOGBUFFER and WRITELOG waits. As a best practice, make sure to split your transaction log and database data files on separate disks, so they do impact each other in times of heavy load. Also monitor the disk the

transaction log is located on using the disk-performance counters in Perfmon, like Avg. Disk sec/write, to show you the write latency, and Disk Writes/sec, to show the write IOPS and check if the values are inside the acceptable range.

If your SQL Server instance is running SQL Server 2014, you could choose to make use of the Delayed Durability option, which was introduced in SQL Server 2014. In short, enabling this option will no longer flush the log buffer content to disk when a transaction commits, but rather will wait until the log buffer is full (60 KB) before flushing the contents to the transaction log. By enabling this option, you are running a risk that transactions that have committed, but have not yet been written to the transaction log, will be lost during a failure since they will only be written to the transaction log when the log buffer is full.

LOGBUFFER and WRITELOG Summary

Both the LOGBUFFER and WRITELOG wait types are related to the way SQL Server processes transactions. The WRITELOG wait type occurs every time a log record is written to the transaction log and generally isn't a cause for concern. When paired with high LOGBUFFER wait times, high WRITELOG wait times can indicate transaction log pressure. To lower those wait times try to avoid cursor and WHILE statements, since the statements inside the cursor or WHILE clauses will often get implicitly committed, creating a large amount of small writes. Also check your storage configuration to make sure the transaction log isn't on the same drive as the database data file. If high wait times still occur, analyze the performance of the disk on which your transaction log is located.

RESOURCE_SEMAPHORE

The RESOURCE_SEMAPHORE wait type is a memory-related wait type that can show itself when a query-memory request cannot be granted immediately. These waits can occur on servers that are experiencing memory pressure, or when a great number of concurrent queries request memory for expensive operations like sorts or joins.

What Is the RESOURCE_SEMAPHORE Wait Type?

When a query is executed in SQL Server, a series of steps occur before the actual execution. The first step involved is that a compiled plan is generated. This plan contains the logical instructions, or operations, needed to fulfill the query requests. During the

generation of the compiled plan, a calculation is performed to determine the amount of memory needed to execute the query, which depends on the operations involved in the compiled plan. Some of the operations that require memory are sorts and joins, which temporarily store row data in the memory of the SQL Server. The minimum amount of memory needed to perform these sorts or joins is known as the *required memory*, without which the query simply cannot get executed. If more memory is needed to store row data in memory during a sort, for instance, it will be calculated as *additional memory*. Without this additional memory, a query can still get executed, but instead of writing the temporary row data to memory, it will write it to disk.

When the query gets executed, a memory grant will be determined based on the required and additional memory values calculated in the compiled plan. This memory grant is needed in order to perform a memory reservation at an internal object called the resource semaphore. The resource semaphore is responsible for reserving the memory a query needs for execution, but it also manages memory throttling when too many queries concurrently ask for memory reservations or when there is not enough memory available at that time. It does this by maintaining a queue of queries that are requesting memory. If there are no queries inside the queue and a new query requests memory, the resource semaphore will grant it to the query (if enough free memory is available). However, if there is a queue the new query will be put at the end of the queue, and it has to wait for its turn to receive a memory grant.

Before the resource semaphore will grant the requested memory to a query, it will check whether there is enough free memory to execute it. If, for some reason, there is less memory available than the amount requested by the query, the query will be put in the queue again until enough memory is available. When a query is inside the resource semaphore queue waiting for its requested memory, the time it spends inside the queue will be recorded as the RESOURCE_SEMAPHORE wait type.

There is a maximum amount of memory available for the resource semaphore to use, and it is allocated from the buffer cache. The resource semaphore can allocate up to 75% of the memory from the buffer cache for memory grants, but a single query can never get more than 25% of that amount. For instance, if we have a SQL Server with a buffer cache that can grow to 500 MB, we would have a maximum of 375 MB for memory grants. A single query in this example can never receive more than 93 MB. Having that much memory being possibly granted to queries can be problematic, since that memory is not being used for the buffer cache, meaning more IOs to the storage subsystem are needed to retrieve and write data pages.

RESOURCE_SEMAPHORE Example

For this example we are going to execute a query against the AdventureWorks database that involves a sort operation. As I mentioned in the previous section, a query that involves a sort will request memory from the resource semaphore so as to perform the sort operation. We will also use the Ostress tool to create a situation where multiple queries are requesting memory, creating a queue at the resource semaphore.

Let's take a look at the query and the memory grant information that we will be executing in Listing 6-8.

Listing 6-8. Sort query against the AdventureWorks database

```
SELECT
    SalesOrderID,
    SalesOrderDetailID,
    ProductID,
    CarrierTrackingNumber
FROM
    Sales.SalesOrderDetail
ORDER BY CarrierTrackingNumber ASC
```

As you can see, this is a relatively simple query that returns some information from the Sales.SalesOrderDetail table, ordered by the CarrierTrackingNumber.

If we enable the Include Actual Execution Plan option and execute the query, we can take a look at the amount of memory that was needed to execute it. The results on my test SQL Server are shown in Figure 6-19. You can access these properties by showing the Properties window (View ➤ Properties Window) or by pressing F4 and selecting the SELECT operator.

MemoryGrantInfo	
DesiredMemory	13568
GrantedMemory	13568
GrantWaitTime	0
MaxQueryMemory	416584
MaxUsedMemory	8608
RequestedMemory	13568
RequiredMemory	512
SerialDesiredMemory	13568
SerialRequiredMemory	512

Figure 6-19. MemoryGrantInfo inside the properties of the execution plan

Because we requested the actual execution plan, we can also see the amount of memory that was granted to the query for execution. In this case the query got 13,568 KB (13.5 MB) granted by the resource semaphore, as shown in the GrantedMemory property. The minimal amount of memory needed to execute the query, the required memory, was 512 KB, shown by the RequiredMemory property. The query asked for 13,568 KB, as shown in the DesiredMemory property, which is the sum of the required and additional memory. We can see the query received what it asked for, since both the GrantedMemory and DesiredMemory have the same value.

There are two other properties in Figure 6-19 I would like to point out, the SerialDesiredMemory and the SerialRequiredMemory properties. In the case of this query, both these properties have the same values as the DesiredMemory and RequiredMemory properties. This is because the query was performed without using parallelism. When you use parallelism in your queries, more memory is needed to perform the sort operation since work is split up among threads. Figure 6-20 shows the MemoryGrantInfo properties when I forced the query in Listing 6-8 to use parallelism, spreading the work among four threads.

MemoryGrantInfo	
DesiredMemory	15488
GrantedMemory	15488
GrantWaitTime	0
MaxUsedMemory	9024
RequestedMemory	15488
RequiredMemory	2432
SerialDesiredMemory	13568
SerialRequiredMemory	512

Figure 6-20. MemoryGrantInfo properties when executing a parallel query

As you can see, the SerialRequiredMemory has the same value as when we executed the query serially. The RequiredMemory and RequestedMemory have increased in size so that the sort operation can be completed using parallelism. You should keep this information in mind when you run into memory-related issues and when many of your queries involve sort and join operations that are performed using parallelism, since parallelism simply requires more memory.

Now that we know how much memory is needed to execute the query in Listing 6-8, let's use Ostress to execute the query using multiple connections. Before I start Ostress, I must change the maximum server memory value to 250 MB using the following query:

```
EXEC sys.sp_configure N'max server memory (MB)', N'250'
GO
RECONFIGURE WITH OVERRIDE
GO
```

I saved the query in Listing 6-8 to a .sql file named resource_semaphore.sql and executed Ostress using the following command line: "C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E -dAdventureWorks -i"C:\resource_semaphore.sql" -n20 -r1 -q

This will execute the resource_semaphore.sql script against the AdventureWorks database with 20 concurrent connections, with each connection performing the query one time.

While Ostress is running, I query the sys.dm_os_waiting_tasks DMV, looking for RESOURCE_SEMAPHORE waits; some of the results are shown in Figure 6-21.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type
4	0x000000825B7A8CA8	88	0	4736	RESOURCE_SEMAPHORE
5	0x000000825B7A8CA8	88	0	4736	RESOURCE_SEMAPHORE
6	0x00000082504D2108	90	0	4736	RESOURCE_SEMAPHORE
7	0x00000082504D2108	90	0	4736	RESOURCE_SEMAPHORE
8	0x0000008251E89848	91	0	4736	RESOURCE_SEMAPHORE
9	0x0000008251E89848	91	0	4736	RESOURCE_SEMAPHORE
10	0x000000825B7A9848	93	0	4736	RESOURCE_SEMAPHORE
11	0x000000825B7A9848	93	0	4736	RESOURCE_SEMAPHORE

Figure 6-21. RESOURCE_SEMAPHORE waits in the sys.dm_os_waiting_tasks DMV

Because we set our maximum server memory to 250 MB, and each query requests 13.25 MB memory, we do not have enough memory free to grant all the memory requested. This will result in the RESOURCE_SEMAPHORE wait type you can see in Figure 6-21.

There are various other resources we can use to analyze RESOURCE_SEMAPHORE waits. The resource semaphores themselves have their own DMV, `sys.dm_exec_query_resource_semaphores`, which will return information about their memory consumption and outstanding and waiting grants. Figure 6-22 shows the results of the query that follows against the `sys.dm_exec_query_resource_semaphores` DMV, while running the Ostress workload:

```
SELECT
    target_memory_kb,
    max_target_memory_kb,
    total_memory_kb,
    available_memory_kb,
    granted_memory_kb,
    grantee_count,
    waiter_count
FROM sys.dm_exec_query_resource_semaphores
WHERE pool_id = 2
```

I am filtering out pool_id 1 because this pool will not handle user queries.

	target_memory_kb	max_target_memory_kb	total_memory_kb	available_memory_kb	granted_memory_kb	grantee_count	waiter_count
1	40360	187240	40360	13224	27136	2	18
2	5120	NULL	5120	5120	0	0	0

Figure 6-22. `sys.dm_exec_query_resource_semaphores`

As you might have noticed, two rows are returned. This is because there are actually two different resource semaphores. The top row is the “regular” resource semaphore. This will handle queries that request more than 5 MB memory. The second row (identified by the NULL value of the `max_target_memory_kb` column) returns information for the “small” resource semaphore, which handles queries that are smaller than 5 MB. Because our query requested more than 5 MB of memory, we will receive our memory grants from the regular resource semaphore.

Let's go through the various columns that are returned by the query against the sys.dm_exec_query_resource_semaphore DMV:

- The target_memory_kb column returns the amount of memory in KB that this resource semaphore plans to use as a maximum amount of memory it can grant to queries.
- The max_target_memory_kb column returns the maximum amount of memory this resource semaphore could grant.
- The total_memory_kb column returns the total memory held by the resource semaphore and is the sum of the available_memory_kb and the granted_memory_kb.
- The granted_memory_kb returns the amount of memory that is granted to queries at this time.
- The grantee_count and waiter_count columns return the amount of grants that have currently been satisfied or are waiting in the resource semaphore queue.

From this information we can see that the information returned by the granted_memory_kb column is correct, and that our test queries are requesting the memory grants. We know from the execution plan that our test query will request 13,568 KB. Since the grantee_count column shows us that two memory requests are granted, we can multiply the amount of memory requests with the amount of memory per query ($2 \times 13,568$ KB), which ends up being 27,136 KB, the amount of granted memory in Figure 6-22.

We can also use Perfmon to monitor the total size of the granted memory by looking at the SQLServer:Memory Manage\Granted Workspace Memory (KB) counter, as shown in Figure 6-23.

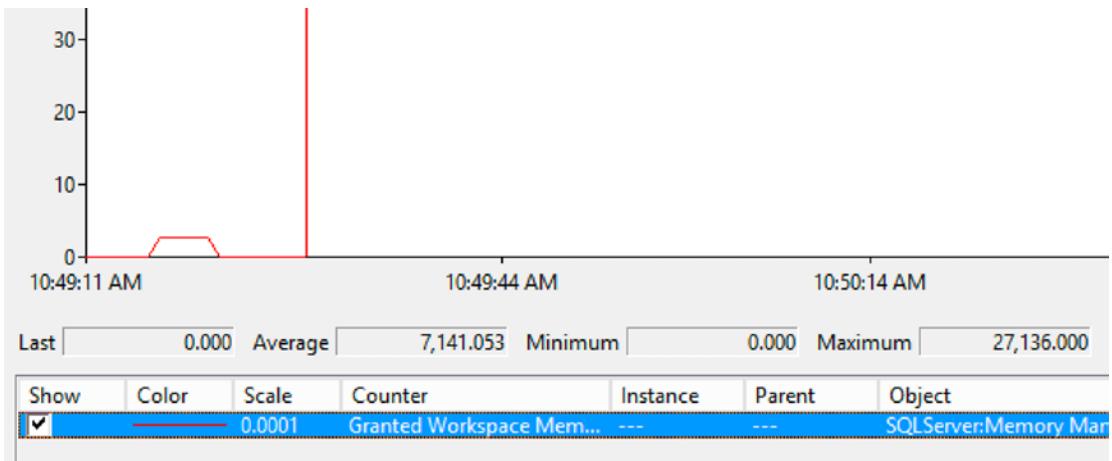


Figure 6-23. Granted Workspace Memory (KB) Perfmon counter

Notice the spike in Figure 6-23, which occurred when I executed the Ostress workload, and just as in the results in Figure 6-22, the amount of memory granted was 27,136 KB.

Lowering RESOURCE_SEMAPHORE Waits

There are various possible methods you can use to lower, or even resolve, the wait times of the RESOURCE_SEMAPHORE wait type. The most obvious one would be adding more memory, but this can end up being an expensive solution while other less expensive options exist.

The first possible solution would be to look at the queries that are requesting large amounts of memory for their execution. You should focus on the queries that are performing large sorts or joins (especially hash joins) and check whether you can lower the number of rows that need to be sorted or joined, or avoid the sort or join completely. One way to avoid a sort operation would be to add an index to the table where the sort is performed. If the order of values inside the index were the same as the sort operation, a sort operation would no longer be necessary, since the index would already have ordered the results.

Another solution involves parallelism. If queries use parallelism during sort or join operations, more memory is requested than when the query is executed serially. Modifying queries so they don't use parallelism, by either using query hints or changing the parallelism configuration for the whole SQL Server instance, will result in lower amounts of memory being required to execute the queries.

Finally, if you are running an Enterprise Edition of SQL Server, you could use the resource governor feature to configure the memory usage of each resource pool. By configuring the amount of memory a certain resource pool can use, you can also set the amount of memory a resource semaphore can grant. We won't go into detail about the resource governor feature in this book, but more information can be found on the MSDN page of the resource governor with the following hyperlink:

<https://msdn.microsoft.com/en-us/library/bb933866.aspx>.

RESOURCE_SEMAPHORE Summary

The RESOURCE_SEMAPHORE wait type is related to the amount of memory a query needs to perform certain operations, like sorts and joins. An object, named a resource semaphore, is responsible for managing and throttling the memory requests of queries. If a query requests more memory than the resource semaphore can grant, the memory request will be moved into the resource semaphore queue. While the memory request is inside the resource semaphore queue, RESOURCE_SEMAPHORE wait times are recorded. There are various methods to lower or resolve RESOURCE_SEMAPHORE waits. You can choose to add more memory to the SQL Server instance or optimize the queries so sorts and joins do not require as much memory. Another option is using the resource governor, where you can define resource pools so as to minimize the impact of large memory requests.

RESOURCE_SEMAPHORE_QUERY_COMPILE

In the previous section we discussed the RESOURCE_SEMAPHORE wait type, which indicates that there is not enough free memory available for certain query operations like sorts and joins. Just like the RESOURCE_SEMAPHORE wait type, the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type is also related to the memory of your SQL Server instance. But instead of indicating a shortage in query memory, the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type indicates a memory shortage during the compilation process of the query.

What Is the RESOURCE_SEMAPHORE_QUERY_COMPILE Wait Type?

In the explanation of the RESOURCE_SEMAPHORE wait type, we discussed what resource semaphores are and what they do. For the explanation of the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type, we are going to dive a little deeper into the inner workings of the resource semaphore.

You should think of resource semaphores as “gateways” that throttle direct access to memory resources. There are different tasks a resource semaphore can do. In the previous section we discussed the resource semaphores that were responsible for granting memory for certain operations, like sorts and joins. We also noted that there are two resource semaphores that are responsible for granting this memory—the regular resource semaphore that handles queries that request 5 MB or more memory and the small resource semaphore that handles memory grants for queries that request less than 5 MB memory.

The resource semaphores that are related to the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type are responsible for memory grants that are needed during the compilation process of a query, excluding the memory needed for query execution. Just like the resource semaphores in the previous section, the ones responsible for memory grants during the compilation process also have different gateways. Figure 6-24 shows the different gateways for the compilation-memory resource semaphore.

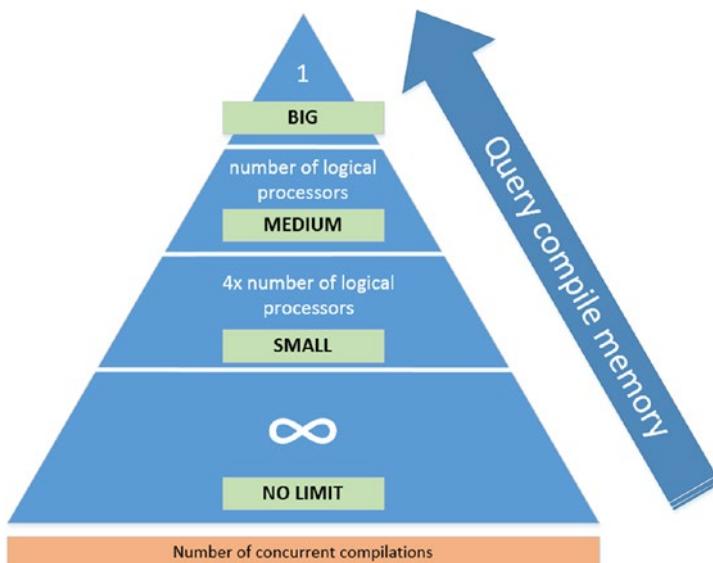


Figure 6-24. Compilation-memory resource semaphore

By default there are three gateways: small, medium, and big. Depending on the amount of memory the compilation of a query needs, it will get assigned to either of the three. If the amount of memory needed for compilation is less than the memory threshold for the small gateway, the query does not have to pass through a gateway. The amount of concurrent compilations, or queries, that can move through the gateway simultaneously is calculated by the number of logical processors available for your SQL Server instance. For example, if your SQL Server instance has four logical processors, the small gateway will allow 16 concurrent compilations and the medium gateway 4. The big gateway will always only allow one query at a time to compile.

The memory threshold for the small gateway is static, but for the medium and big gateways the thresholds are dynamic. This means that the compilation memory needed to reach the medium or big gateways can change during the runtime of your SQL Server instance.

The whole purpose of these gateways is to ensure that the need for compilation memory stays under control. This avoids out-of-memory situations in cases where many large compilation-memory requests would automatically be granted and would drain the SQL Server instance of its memory.

Before we continue and take a look how we can access gateway information from inside SQL Server, let's go through an example of a query compilation.

Say we have a query that needs 1560 KB of compilation memory. The query will start by requesting a gateway. Since our small gateway has a threshold of 370 KB and our medium gateway has a threshold of 5346 KB, the query will end up in the small gateway. If there are any queries currently in a queue at the small gateway, the query will enter the queue and wait until its turn, all the time logging RESOURCE_SEMAPHORE_QUERY_COMPILE wait time. While the query is getting compiled, the amount of memory used during the compilation is tracked; if the query ends up using more memory and reaches the threshold of the medium gateway, it will get moved to the medium gateway. When the query compilation is finished, it will be removed from the gateway.

We can access information about the resource semaphore gateways from inside SQL Server by executing the DBCC MEMORYSTATUS command. Somewhere in the enormous amount of results you will find the gateway information as shown in Figure 6-25.

	Small Gateway (internal)	Value
1	Configured Units	8
2	Available Units	8
3	Acquires	0
4	Waiters	0
5	Threshold Factor	380...
6	Threshold	380...

	Medium Gateway (internal)	Value
1	Configured Units	2
2	Available Units	2
3	Acquires	0
4	Waiters	0
5	Threshold Factor	12
6	Threshold	-1

Figure 6-25. Gateway information returned by the DBCC MEMORYSTATUS command

Let's go through the results that are returned for the gateways.

The Configured Units row returns the maximum amount of concurrent compilations allowed for this gateway. This is determined by the number of logical processors available for your SQL Server instance. Because my test SQL Server has two logical processors, I have eight slots for the small gateway ($4 \times$ logical number of processors) and two for the medium gateway. The Available Units row shows the number of currently free slots for this gateway, while the Acquires row shows the slots currently taken by compilations. The number of queries that have to wait for a free slot are shown in the Waiters row. The Threshold value is the amount of memory in bytes that a query compilation would need in order to enter the gateway. For my test SQL Server system, the small gateway has a threshold of 380,000 bytes, or 371 KB. As you might notice in Figure 6-25, the medium gateway has a threshold of -1. This is because of the dynamic nature of the thresholds of the medium and big gateways. Since there is no activity at the gateway below the medium one, there is no need to set a threshold yet.

RESOURCE_SEMAPHORE_QUERY_COMPILE Example

To show you an example of RESOURCE_SEMAPHORE_QUERY_COMPILE waits in action, I am going to execute the query in Listing 6-9 multiple times, using many concurrent connections. The query is a dynamic query that selects a random row from two joined tables inside the AdventureWorks database. In this case it doesn't matter if

any results are returned or not—the thing we are trying to achieve here is the creation of compilation-memory contention.

Listing 6-9. RESOURCE_SEMAPHORE_QUERY_COMPILE wait query

```
DECLARE @ID VARCHAR(250)
DECLARE @SQL VarChar(MAX)
SET @ID = FLOOR(RAND()*(20000-1)+1);

SET @SQL =
'

SELECT
    ' + @ID + ',
    SUM(soh.SubTotal),
    COUNT(soh.SubTotal)
FROM sales.SalesOrderHeader soh
INNER JOIN person.Person p
    ON soh.SalesPersonID = p.BusinessEntityID
WHERE p.BusinessEntityID = ' + @ID +
'

EXEC (@SQL)
```

Before we execute the query with many concurrent connections, let's check how much compilation memory would be needed. We can do this by executing the query in Listing 6-9 in SQL Server Management Studio and enabling the actual execution plan.

After executing the query and opening the actual execution plan, we need to look at the `CompileMemory` property. You can access these properties by showing the Properties window (View ▶ Properties Window) or by pressing F4 and selecting the SELECT operator. Figure 6-26 shows the actual execution plan properties on my Test SQL Server.

Misc	
Cached plan size	32 KB
CompileCPU	6
CompileMemory	408
CompileTime	102
Degree of Parallelism	0

Figure 6-26. Miscellaneous execution plan properties

The value returned by the `CompileMemory` property is the amount of compile memory needed expressed in KB. For this query 408 KB is needed for compilation. The threshold for the small gateway on my Test SQL Server was 371 KB, so I am pretty sure the query will access the small gateway.

Again, we are going to use the `Ostress` utility to generate the needed concurrent connections to execute the query. I saved the query to the `resource_semaphore_compile.sql` file and then used that file as input for the following `Ostress` command. Because the query is very fast, I let every connection execute it 100 times so that we have some time to look at the wait statistics.

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -i"C:\resource_semaphore_compile.sql" -n200 -r100 -q
```

After a few seconds many `RESOURCE_SEMAPHORE_QUERY_COMPILE` waits can be seen in the `sys.dm_os_waiting_tasks` DMV, as shown in Figure 6-27.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type
1	0x00000082570888C8	288	0	215	RESOURCE_SEMAPHORE_QUERY_COMPILE
2	0x00000082570888C8	288	0	215	RESOURCE_SEMAPHORE_QUERY_COMPILE
3	0x00000082570888C8	288	0	215	RESOURCE_SEMAPHORE_QUERY_COMPILE
4	0x000000825A9C48C8	167	0	212	RESOURCE_SEMAPHORE_QUERY_COMPILE
5	0x000000825A9C48C8	167	0	212	RESOURCE_SEMAPHORE_QUERY_COMPILE
6	0x000000825A9C48C8	167	0	212	RESOURCE_SEMAPHORE_QUERY_COMPILE
7	0x000000825DD01C28	92	0	169	RESOURCE_SEMAPHORE_QUERY_COMPILE
8	0x000000825DD01C28	92	0	169	RESOURCE_SEMAPHORE_QUERY_COMPILE
9	0x000000825DD01C28	92	0	169	RESOURCE_SEMAPHORE_QUERY_COMPILE
10	0x000000825A9C4CA8	211	0	204	RESOURCE_SEMAPHORE_QUERY_COMPILE
11	0x000000825A9C4CA8	211	0	204	RESOURCE_SEMAPHORE_QUERY_COMPILE

Figure 6-27. `RESOURCE_SEMAPHORE_QUERY_COMPILE` waits

If we now execute the `DBCC MEMORYSTATUS` command we should be able to find out at what gateway the compilation contention is occurring. Figure 6-28 shows the gateway output of the `DBCC MEMORYSTATUS` command on my test SQL Server.

	Small Gateway (default)	Value
1	Configured Units	8
2	Available Units	0
3	Acquires	6
4	Waiters	22
5	Threshold Factor	380000
6	Threshold	380000

	Medium Gateway (default)	Value
1	Configured Units	2
2	Available Units	2
3	Acquires	0
4	Waiters	0
5	Threshold Factor	12
6	Threshold	7041820

Figure 6-28. DBCC MEMORYSTATUS during compilation contention

As you can see in Figure 6-28, if we look at the number of Available Units, there are no available slots left for new compilation-memory requests. As a matter of fact, we have 22 compilation-memory requests waiting in the resource semaphore queue. Also note that the threshold of the medium gateway has now changed from -1 to 7,041,820 bytes (6876 KB). Now that contention is occurring on a lower gateway, the threshold for the medium gateway is dynamically determined, even though there are no compilation-memory requests being processed by this gateway.

Lowering RESOURCE_SEMAPHORE_QUERY_COMPILE Waits

The methods you can use to lower the wait times of the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type are in many cases the same as those that you would use to lower or resolve RESOURCE_SEMAPHORE waits. Just like the RESOURCE_SEMAPHORE wait type, the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type is memory related, so if you can increase the total amount of memory available for query compilation, chances are you will lower or resolve RESOURCE_SEMAPHORE_QUERY_COMPILE wait times. Increasing memory is, however, in many cases the last resort.

Because we can access very specific information about the gateways of the resource semaphore that is dealing with the compilation memory by using the DBCC MEMORYSTATUS command, a good first step is to analyze the usage patterns of the gateways. If you notice that one specific gateway constantly has waiting memory requests, then the memory threshold of that gateway, or the maximum allowed amount

of concurrent compilation-memory requests, should give you some hints about the root cause. For instance, if you notice many queued compilation-memory requests at the big gateway (which only allows one query at a time), the source of your RESOURCE_SEMAPHORE_QUERY_COMPILE wait times may be the queries that request a large amount of compilation memory. Another cause may be a large number of concurrent queries that all need to access the small gateway, which was the case in our example, causing a queue at the gateway.

In these cases you should find the specific queries that cause the queues at the gateways and try to optimize them, either by lowering the amount of compilation memory or by making sure fewer compilations happen. The latter can be done by making sure your queries are being parameterized correctly. Queries that generate ad hoc plans every time they are executed can be a cause of RESOURCE_SEMAPHORE_QUERY_COMPILE waits, especially if they are executed very frequently and concurrently. Jonathan Kehayias of SQLskills has written an excellent blog post on how you can query the plan cache to detect heavy compilation queries; it can be found at www.sqlskills.com/blogs/jonathan/identifying-high-compile-time-statements-from-the-plan-cache/. Using the script in Jonathan's blog post should help you with detecting those queries that require a large amount of compilation memory.

If your SQL Server is under memory pressure, it is also possible to see RESOURCE_SEMAPHORE_QUERY_COMPILE waits occur. This happens because of the dynamic compilation-memory thresholds of the medium and big gateways. If SQL Server is under memory pressure, the thresholds of both these gateways will lower, giving more queries the chance to use the medium or big gateways. But because the medium and big gateways allow fewer concurrent compilations, in the small gateway the available concurrent slots will be filled faster.

Just as with the RESOURCE_SEMAPHORE wait type, you can use the resource governor to split workloads into specific resource pools. Each resource pool will have its own resource semaphores responsible for granting compilation memory, making it possible to split heavy compilation-memory usage across multiple resource pools.

RESOURCE_SEMAPHORE_QUERY_COMPILE Summary

Just like the resource semaphores that are needed to grant memory requests for specific query operations, resource semaphores exist for access to compilation memory. These resource semaphores throttle access to compilation memory through the usage of gateways. When a query is compiled, it will approach a gateway based on the amount

of compilation memory it needs. The gateway can then grant the compilation memory requested or put the request in a queue if there are more requests that concurrently want to access the gateway. When a query is waiting inside one of these queues, the RESOURCE_SEMAPHORE_QUERY_COMPILE wait type is recorded.

Resolving or lowering RESOURCE_SEMAPHORE_QUERY_COMPILE wait times is commonly achieved by either freeing up more memory or by lowering the compilation-memory needs of queries.

SLEEP_BPOOL_FLUSH

The SLEEP_BPOOL_FLUSH wait type is directly related to the checkpoint process inside SQL Server. The checkpoint process is responsible for writing modified, or “dirty,” data pages from the buffer pool to the database data file on disk. So next to having a close relationship with the checkpoint process, SLEEP_BPOOL_FLUSH waits also have a relationship with the performance of your storage subsystem. If we search for the definition of the SLEEP_BPOOL_FLUSH wait type on Books Online, Microsoft describes the wait type as occurring “when a checkpoint is throttling the issuance of new I/Os in order to avoid flooding the disk subsystem.”

It is pretty common to see SLEEP_BPOOL_FLUSH waits occur, and frequently they will not indicate a problem. There are, however, cases where SLEEP_BPOOL_FLUSH waits can indicate performance problems that are related to either the checkpoint process or the storage subsystem.

What Is the SLEEP_BPOOL_FLUSH Wait Type?

To get a better understanding of how the SLEEP_BPOOL_FLUSH wait type gets recorded, we need an understanding of how the checkpoint process works inside SQL Server.

The checkpoint process is an internal SQL Server process that is responsible for writing modified (dirty) pages from the buffer cache to the database data file. One of the main reasons for this is to speed up recovery of your database when an unexpected failure occurs. When an unexpected failure occurs, SQL Server needs to go back to the state that existed before the failure. It will do this by using the contents of the transaction log to redo, or undo, changes that were made to data pages. If the data page was modified, but the change was not yet written to the database data file, SQL Server will need to redo the change to the data page. If a checkpoint already wrote the changed

data page to the database data file, this step is not needed, which speeds up the recovery process for the database because SQL Server knows the data was written to the database data file. Figure 6-29 shows the (simplified) process that happens when a data page gets modified.

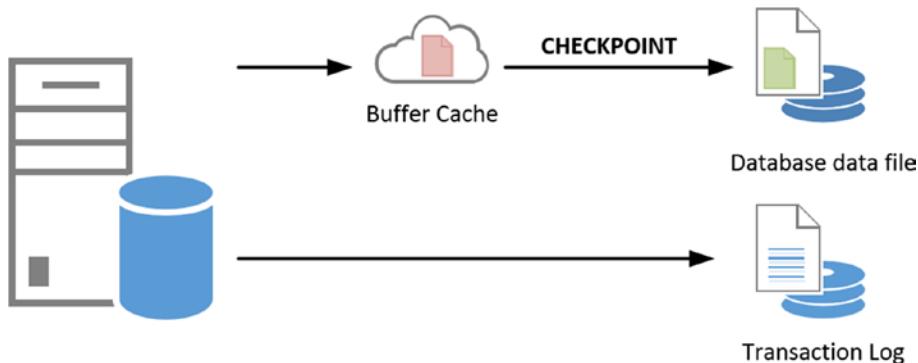


Figure 6-29. Data modification process

The first thing that happens when a data page is modified by a committed transaction is that the change will be recorded in the transaction log (first in the log buffer then to disk as described in the WRITELOG and LOGBUFFER wait types section). The modification of the data page will happen in the buffer cache, and the data page will be marked as dirty (red page icon). When a checkpoint occurs, which can be for multiple reasons as we will discuss later, all of the data pages that have been marked as dirty since the previous checkpoint will be written to the physical database data file on your storage subsystem, regardless of the state of the transaction that created those dirty pages (green page icon).

The checkpoint process is executed by SQL Server automatically roughly once every minute, which is the default recovery time interval if you are a lower version of SQL Server than SQL Server 2016. This does not mean that a checkpoint will occur every minute exactly. The values you can specify for the recovery interval are the upper time limit at which a checkpoint should occur, the checkpoint process analyses of the outstanding I/O requests, and latency; throttle checkpoint operations to avoid overloading the storage subsystem.

The following list will describe the various checkpoint types available in SQL Server:

- The internal checkpoint type is not configurable and occurs automatically when certain actions are performed; for instance, a database backup.
- Automatic: These are the default checkpoints, on SQL Server version lower than 2016, that occur roughly every minute when left at their default value of 0. We can change the interval of the checkpoint process by changing the recovery interval configuration option under the Server Properties ► Database Settings page in SQL Server Management Studio. We can only change it to a value in minutes, and it will be used for all databases inside the SQL Server instance.
- Manual: You can manually cause checkpoints to occur by issuing the CHECKPOINT T-SQL command. Optionally, you can specify the time in seconds at which the checkpoint must be completed. If you do issue a manual checkpoint, it will run in the context of the current database. For example, executing CHECKPOINT 10 in a query window will perform a checkpoint within 10 seconds of the time you executed the query.
- Indirect: SQL Server 2012 added an extra option to configure checkpoint intervals on a per-database level. Configuring this option to a value greater than the default 0 will overwrite the automatic checkpoint process for the specific database. You can use indirect checkpoints for a specific database by using the following command:
`ALTER DATABASE [db name] SET TARGET_RECOVERY_TIME = [time in seconds or minutes].`
- With the release of SQL Server 2016 Indirect Checkpoint became the new default setting of the Checkpoint process (with the value of 60).

As I mentioned before, SQL Server will attempt to throttle the checkpoint process to avoid overloading the storage subsystem if it believes this is necessary. It monitors the number of outstanding requests to the storage subsystem and tries to detect if there is any latency. Using this information, it will throttle the amount of IOs the checkpoint process generates so as to avoid a too-heavy load on the storage subsystem. When the checkpoint process is getting throttled, the SLEEP_BPOOL_FLUSH wait type will be recorded.

SLEEP_BPOOL_FLUSH Example

The following example shows the impact of the SLEEP_BPOOL_FLUSH wait type on SQL Server versions lower than SQL Server 2016. As mentioned earlier, in SQL Server 2016, the way SQL Server handles the Checkpoint process has changed which means it is far less likely for the wait type to show up in an example like the following.

Generating SLEEP_BPOOL_FLUSH waits is relatively simple, and the script in Listing 6-10, which is almost the same one as we used for the LOGBUFFER and WRITELOG wait types, will put pressure on the checkpoint process such that SLEEP_BPOOL_FLUSH waits will occur.

Listing 6-10. Generate SLEEP_BPOOL_FLUSH waits

```
USE trans_demo
GO

DECLARE @i INT
SET @i = 1

WHILE @i < 100
    BEGIN
        INSERT INTO transactions
            (t_guid)
        VALUES
            (newid())
        SET @i = @i + 1
        -- Force a checkpoint to occur within 1 second
        CHECKPOINT 1
    END
```

Since we are also using the same database as in the LOGBUFFER and WRITELOG wait types example, Listing 6-11 shows the script to create the database if it doesn't exist already.

Listing 6-11. Create trans_demo database

```
USE master
GO

-- Create demo database
CREATE DATABASE [trans_demo]
ON PRIMARY
(
    NAME = N'trans_demo', FILENAME = N'D:\Data\trans_demo.mdf' ,
    SIZE = 153600KB , FILEGROWTH = 10%
)
LOG ON
(
    NAME = N'trans_demo_log', FILENAME = N'D:\Log\trans_demo.ldf' ,
    SIZE = 51200KB , FILEGROWTH = 10%
)
GO

-- Make sure recovery model is set to full
ALTER DATABASE [trans_demo] SET RECOVERY FULL
GO

-- Perform full backup first
-- Otherwise FULL recovery model will not be affected
BACKUP DATABASE [trans_demo]
TO DISK = N'F:\Backup\trans_demo_Full.bak'
GO

-- Create a simple test table
USE trans_demo
GO

CREATE TABLE transactions
(
    t_guid VARCHAR(50)
)
GO
```

What the script in Listing 6-10 will do is perform an insert of a random GUID into the transactions table inside a loop that is executed 100 times. Every time it enters a new GUID, it will issue a CHECKPOINT command with a time limit of 1 second. This forces the checkpoint process to perform a checkpoint within the 1-second time limit.

Before running the script in Listing 6-10, I cleared the sys.dm_os_wait_stats DMV using the DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR) command.

After almost 70 seconds the script completed on my test SQL Server. I then executed the following query to take a look at the SLEEP_BPOOL_FLUSH wait times:

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'SLEEP_BPOOL_FLUSH';
```

The results of the query can be seen in Figure 6-30.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SLEEP_BPOOL_FLUSH	311	60415	938	26

Figure 6-30. SLEEP_BPOOL_FLUSH waits

As you can see, the SLEEP_BPOOL_FLUSH wait time has a very high amount of wait time after running the script in Listing 6-10. Normally you would expect those wait times to be either very low or close to zero. If we were to remove the CHECKPOINT command from the script completely and let SQL Server decide on when to run the checkpoint process, we not only get a completely different result, as shown in Figure 6-31, but also the script's runtime is decreased to just a few milliseconds.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SLEEP_BPOOL_FLUSH	0	0	0	0

Figure 6-31. SLEEP_BPOOL_FLUSH wait times after removing CHECKPOINT

Lowering SLEEP_BPOOL_FLUSH Waits

Even though it is not very common to run into performance problems caused by the SLEEP_BPOOL_FLUSH wait type, there are various methods to lower the wait times.

The most obvious one would be to check the various configuration options available to manually configure the recovery interval that we discussed earlier. The lower the value of the recovery interval, the more often checkpoint processes will take place, and the bigger the chance of running into SLEEP_BPOOL_FLUSH waits. Also, as you noticed in the example, performing frequent CHECKPOINT commands inside transactions can lead to SLEEP_BPOOL_FLUSH waits.

Another possible cause can be the storage subsystem on which your database data file resides. As explained earlier, the checkpoint process calculates the load of the storage subsystem and then decides if throttling its throughput is needed. If there is a frequent need of throttling because your storage subsystem is busy, you are more likely to see SLEEP_BPOOL_FLUSH waits occur.

If you are running SQL Server 2016, chances are you will never run into very high SLEEP_BPOOL_FLUSH wait times since the default way SQL Server handles the process has been changed.

SLEEP_BPOOL_FLUSH Summary

The SLEEP_BPOOL_FLUSH wait type is closely related to the checkpoint process in SQL Server. The checkpoint process is responsible for writing modified, or dirty, data pages from the buffer cache to the database data file. The checkpoint process analyzes the performance of the storage subsystem before it writes the dirty pages to disk, and if the storage subsystem is busy, the checkpoint process will throttle its throughput, resulting in SLEEP_BPOOL_FLUSH waits. It is not very common to see very high SLEEP_BPOOL_FLUSH wait times, but they can impact performance nonetheless. Queries that frequently execute the CHECKPOINT T-SQL command, or a recovery interval that is configured to a very low value, can be possible causes for seeing SLEEP_BPOOL_FLUSH waits occur. The performance of your storage subsystem can also impact the checkpoint process if it is forced to throttle its throughput.

WRITE_COMPLETION

As with the ASYNC_IO_COMPLETION and IO_COMPLETION wait types, the WRITE_COMPLETION wait type is related to specific actions SQL Server performs on the storage subsystem. Again, it is very normal to see WRITE_COMPLETION waits occur on your SQL Server instance, and they should only be a cause for concern if the wait times are way higher than normal.

What Is the WRITE_COMPLETION Wait Type?

The WRITE_COMPLETION wait type is a relative of the IO_COMPLETION wait type. But where the IO_COMPLETION wait type is logged for specific read and write operations, the WRITE_COMPLETION wait type is only logged for some very specific write operations. Some of these write operations are growing a data or log file or performing the DBCC CHECKDB command.

Since the WRITE_COMPLETION wait type is related to writing SQL Server data to the storage subsystem, the performance of it can have an impact on the wait times.

WRITE_COMPLETION Example

To show you an example of a WRITE_COMPLETION wait occurring, I am going to perform a CHECKDB against the AdventureWorks database after clearing the sys.dm_os_wait_stats DMV using the DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR) command.

Keep in mind that this example is a completely normal situation in which WRITE_COMPLETION waits can occur, and it shouldn't stop you from performing regular database consistency checks!

[Listing 6-12](#) shows the query I executed to generate a few WRITE_COMPLETION waits.

Listing 6-12. Generate WRITE_COMPLETION waits

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
DBCC CHECKDB ('AdventureWorks');
SELECT * FROM sys.dm_os_wait_stats
WHERE wait_type = 'WRITE_COMPLETION';
```

The results of the last query in the batch are shown in Figure 6-32.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WRITE_COMPLETION	2	1	1	1

Figure 6-32. WRITE_COMPLETION waits

As you can see, the amount of wait time is so low that it would not be a cause of any concern. This is also partly due to the fact that the AdventureWorks database is very small and the storage performance of my test machine is very fast. Running CHECKDB against larger databases can result in higher wait times.

Lowering WRITE_COMPLETION Waits

If you see high WRITE_COMPLETION wait times, try to find out what process is generating the waits. In many cases it will be caused by a CHECKDB or database data or log file growth.

One thing worth checking is the instant file initialization option discussed in the ASYNC_IO_COMPLETION section earlier in this chapter. Not using this option can impact the duration of the WRITE_COMPLETION wait time.

Another, far less common cause for a higher WRITE_COMPLETION wait time is when you are experiencing page latch contention on your Page Free Space page (or PFS). The PFS page tracks the amount of free space in data pages. If a process needs to modify the PFS page very frequently, it is possible to see WRITE_COMPLETION waits occur along with many PAGELATCH_UP waits, which we will discuss in Chapter 9, “Latch-Related Wait Types.” To give you an example of such a scenario, consider a high amount of concurrent queries that all create a temporary table, insert a few rows, and remove the temporary table again. In this case the PFS page of the tempdb database needs to get updates very frequently to reflect the creation and removal of the temporary tables.

WRITE_COMPLETION Summary

The WRITE_COMPLETION wait type, just like the ASYNC_IO_COMPLETION and IO_COMPLETION wait types, is related to specific storage-related actions performed by SQL Server. Seeing WRITE_COMPLETION waits is very normal and won’t be cause for concern in many situations. Operations such as CHECKDB and database data or log file growth can cause WRITE_COMPLETION waits.

CHAPTER 7

Backup-Related Wait Types

Backups are a very important part of database administration, and in many cases they are essential for the survival of the company you work for. Data has become so important for businesses that if some disaster causes data to be lost, companies can lose large amounts of money, or even go out of business.

There are many methods we can implement on all levels of your IT infrastructure to make sure no data is lost (or as little as possible) during a disaster. We could implement a SAN to make sure our data is not stored on a local disk drive on your server. Or we could design SQL Server AlwaysOn Availability Groups to replicate our data across datacenters. But the first step we should take, and hopefully have already taken, is performing regular backups of our data inside our SQL Server databases.

Implementing and scheduling SQL Server backups is not a very difficult task, and there is no excuse not to perform a backup. The type of backup and the interval of backup operations are dictated by the needs of the organization you work for and are frequently expressed in “RTO” (Recovery Time Objective) and “RPO” (Recovery Point Objective) times. These times represent the amount of time it should take to recover from a disaster and the amount of data loss that is acceptable when a disaster occurs. These two times should be the primary input for your SQL Server backup strategy.

Thankfully, SQL Server has different options available to us for meeting RTO and RPO requirements right out of the box. This means we can use SQL Server’s own backup mechanism to fulfill our company’s RTO and RPO times; we are not necessarily dependent on third-party backup software. Since the SQL Server backup operation is an internal process, there are different wait types associated with it, and in this chapter we will take a look at three of the most common wait types that are directly related to performing backups and restores.

Noticing high wait times on these backup/restore-related wait types will not likely lead to a performance degradation of your SQL Server instance. However, we do have options to optimize the SQL Server backup process that can result in faster backup and restore times. And since backups/restores of your database(s) are vital for the survival of your company, optimizing backup and restore throughput can be well worth the effort.

BACKUPBUFFER

The first backup-related wait types we will discuss is BACKUPBUFFER. If we look up the definition of this wait type on Books Online we would get the following text: “Occurs when a backup task is waiting for data, or is waiting for a buffer in which to store data. This type is not typical, except when a task is waiting for a tape mount.” Apparently, we would only see this wait type when we are writing our backups to a tape device, and this is wrong. BACKUPBUFFER waits will practically always be logged during a backup operation, no matter the destination of the backup file. The reason for this is in the way the SQL Server backup operation uses buffers to read data from the database and write it to the backup file.

What Is the BACKUPBUFFER Wait Type?

To understand how BACKUPBUFFER waits are generated we have to take a look at the internals of the SQL Server backup process. These internals are mostly the same regardless of the backup method you use (i.e., transaction log, differential or full backup) and as such they will encounter the same wait types.

SQL Server allocates buffers for the backup process. These buffers will be filled with data from your database and will be moved through a backup/restore process in order to get written to the backup file (or vice versa for a restore operation). The buffers are allocated inside the memory of your system, but outside the memory of your buffer cache so as to avoid stealing memory from the buffer cache. The size and the amount of the backup buffers are automatically calculated by SQL Server, but we can configure these values ourselves as parameters of the backup/restore command. Figure 7-1 shows how these backup buffers are ordered and moved through a “reader,” which reads the data from your database or backup file to a buffer, and a “writer,” which writes the data from the buffer to the backup file or database.

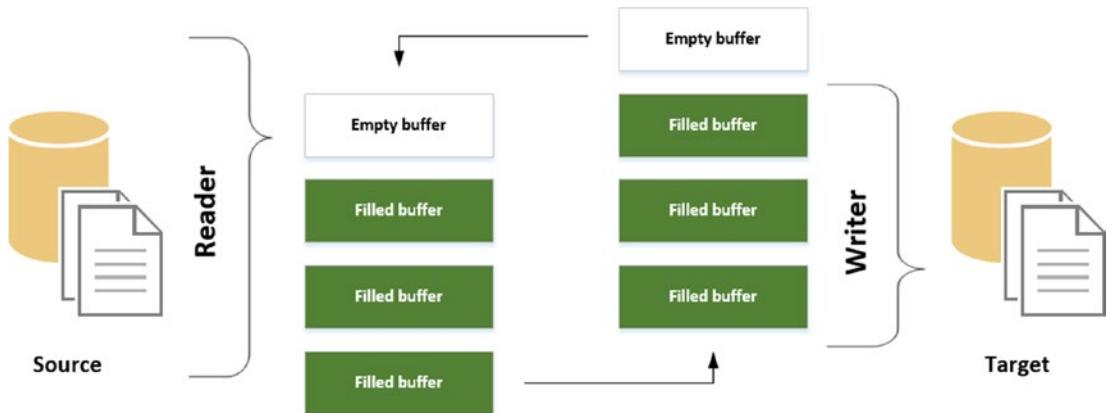


Figure 7-1. Backup buffers moving through reader and writer

We can view information about the buffer amount and size during a backup or restore operation by enabling two trace flags, 3213 and 3605, which will output backup/restore information into the SQL Server error log. The query in Listing 7-1 enables both trace flags and performs a full database backup of the AdventureWorks database on my test SQL Server.

Listing 7-1. Full database backup with backup-information trace flags

```
-- enable trace flags
DBCC TRACEON (3213);
DBCC TRACEON (3605);

-- backup database
BACKUP DATABASE [AdventureWorks]
TO DISK = N'F:\Backup\aw_21042015.bak'
WITH NAME = N'AdventureWorks-Full Database Backup';
GO

-- disable trace flags
DBCC TRACEOFF (3213);
DBCC TRACEOFF (3605);
```

Keep in mind that trace flags inside SQL Server should only be used under the guidance of Microsoft Support. I am enabling them now to show me backup information on my test SQL Server, but I would advise against using them on a production system.

CHAPTER 7 BACKUP-RELATED WAIT TYPES

Inside the SQL Server error log, additional information about the backup we just performed is logged, as you can see in Figure 7-2.

4/21/2015 10:56:57 AM	spid59	DBCC TRACEOFF 3605, server process ID (SPID) 59. This is an informational message only; no user action is required.
4/21/2015 10:56:57 AM	spid59	DBCC TRACEOFF 3213, server process ID (SPID) 59. This is an informational message only; no user action is required.
4/21/2015 10:56:57 AM	Backup	Database backed up. Database: AdventureWorks2012, creation date/time: 2015/04/03(10:51:17), pages dumped: 162
4/21/2015 10:56:47 AM	spid59	Media Buffer size: 1024KB
4/21/2015 10:56:47 AM	spid59	Media Buffer count: 7
4/21/2015 10:56:47 AM	spid59	Flesytem i/o alignment: 512
4/21/2015 10:56:47 AM	spid59	TXF device count: 0
4/21/2015 10:56:47 AM	spid59	Filestream device count: 0
4/21/2015 10:56:47 AM	spid59	Fulltext data device count: 0
4/21/2015 10:56:47 AM	spid59	Tabular data device count: 1
4/21/2015 10:56:47 AM	spid59	Total buffer space: 7 MB
4/21/2015 10:56:47 AM	spid59	Min MaxTransferSize: 64 KB
4/21/2015 10:56:47 AM	spid59	MaxTransferSize: 1024 KB
4/21/2015 10:56:47 AM	spid59	Sets Of Buffers: 1
4/21/2015 10:56:47 AM	spid59	BufferCount: 7
4/21/2015 10:56:47 AM	spid59	Memory limit: 127MB
4/21/2015 10:56:47 AM	spid59	Backup/Restore buffer configuration parameters
4/21/2015 10:56:47 AM	spid59	DBCC TRACEON 3605, server process ID (SPID) 59. This is an informational message only; no user action is required.
4/21/2015 10:56:47 AM	spid59	DBCC TRACEON 3213, server process ID (SPID) 59. This is an informational message only; no user action is required.

Figure 7-2. Additional backup information

In this case, the backup operation created seven buffers, shown by the BufferCount parameter, with a size of 1024 KB each, as shown by the MaxTransferSize parameter. The total memory needed to create the buffers is shown by the Total buffer space parameter, 7 MB (BufferCount * MaxTransferSize). Another interesting bit of information that is returned is the memory limit. This will show the maximum amount of memory outside of the buffer cache that the backup operation could access.

Now that we have an idea of how the backup process works inside SQL Server, let's take a look where the BACKUPBUFFER wait type comes in.

As we described earlier, the SQL Server backup process uses buffers to store data that needs to be written to the backup file. Whenever a buffer is not directly available, the BACKUPBUFFER wait will occur, making the process wait until a full buffer is written to the backup file and it becomes available again.

BACKUPBUFFER Example

Generating BACKUPBUFFER waits is very simple—just perform a backup operation. For this example I ran the query shown in Listing 7-2. The query will first reset the sys.dm_os_wait_stats DMV, then will perform a full backup of the AdventureWorks database, and finally will return the wait statistics information for the BACKUPBUFFER wait type.

Listing 7-2. Generating BACKUPBUFFER waits

```
-- clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- backup database
BACKUP DATABASE [AdventureWorks]
    TO DISK = N'F:\Backup\aw_21042015.bak'
WITH
    NAME = N'AdventureWorks-Full Database Backup';
GO

-- Query BACKUPBUFFER waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'BACKUPBUFFER';
```

The results of the query against the sys.dm_os_wait_stats DMV are shown in Figure 7-3.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	BACKUPBUFFER	18	88	47	2

Figure 7-3. BACKUPBUFFER waits

The total duration of the backup operation was around 1 second on my test SQL Server. Of that 1 second, only 88 milliseconds were spent waiting on free backup buffers.

Lowering BACKUPBUFFER Waits

As stated in the introduction of this chapter, backup-related waits aren't normally any cause for concern since they normally won't impact the performance of your SQL Server instance. However, we can improve backup performance by using the wait statistics information of the various backup-related wait types.

One of the most common ways to lower BACKUPBUFFER wait times is by adding more buffers for the backup operation to use, overwriting the automatic allocation of buffers. We can do this by specifying the BUFFERCOUNT option inside the BACKUP T-SQL command. There is, however, a catch to altering the number of buffers the backup operation can use. Every buffer created will allocate the value of the MAXTRANSFERSIZE option; this value can be automatically calculated by SQL Server itself or by setting the value yourself inside the BACKUP command (up to a maximum of 4,194,304 bytes). Since the backup operation allocates memory outside of the buffer cache, there is a chance that using too many or too large buffers can result in out-of-memory problems. So, be careful when testing what the optimal value for your SQL Server instance is.

Listing 7-3 shows a modification of the query in *Listing 7-2*, which we used to demonstrate BACKUPBUFFER waits occurring. In this case we added the BUFFERCOUNT option and configured it to a value of 200.

Listing 7-3. Database backup with BUFFERCOUNT configured

```
-- clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- backup database
BACKUP DATABASE [AdventureWorks]
TO DISK = N'F:\Backup\aw_21042015.bak'
WITH
NAME = N'AdventureWorks-Full Database Backup',
BUFFERCOUNT = 200;
GO

-- Query BACKUPBUFFER waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'BACKUPBUFFER';
```

The results of the query against the `sys.dm_os_wait_stats` DMV are shown in Figure 7-4.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	BACKUPBUFFER	0	0	0	0

Figure 7-4. BACKUPBUFFER waits

As you can see, the amount of time spent on the BACKUPBUFFER wait type went down to 0 milliseconds instead of the 88 milliseconds it spent when we did not supply the `BUFFERCOUNT` parameter. This happens because the number of buffers we specified were enough to process the backup operation without a need to allocate additional buffers. Since no additional buffers were required, we do not spend time waiting on their allocation.

Another option is to configure the `MAXTRANSFERSIZE` option inside the `BACKUP` T-SQL command. This will allow buffers to be filled with larger units of works, up to a value of 4,194,304 bytes, or 4 MB. Again, allocating more space for the buffers will result in a larger reservation of memory.

BACKUPBUFFER Summary

BACKUPBUFFER waits occur normally during backup or restore operations when the backup/restore operation has to wait for free buffers to become available again. Because they occur normally they shouldn't be a cause for concern. We do have some options for lowering BACKUPBUFFER wait times that will also impact the duration of the backup/restore operation. They should be configured and tested thoroughly though, because setting those parameters too high can result in out-of-memory errors.

BACKUPIO

Just like the BACKUPBUFFER wait type, the BACKUPIO wait type occurs when a part of the backup or restore operation runs into contention problems. Another similarity is the description of this wait type on Books Online: "Occurs when a backup task is waiting for data, or is waiting for a buffer in which to store data. This type is not typical, except when a task is waiting for a tape mount." Again, this wait type is common when performing a backup or restore operation, even when the backup target, or restore source, is not a tape device.

What Is the BACKUPIO Wait Type?

To better understand how BACKUPIO waits are generated, we have to take a look at Figure 7-5, which we showed earlier as Figure 7-1.

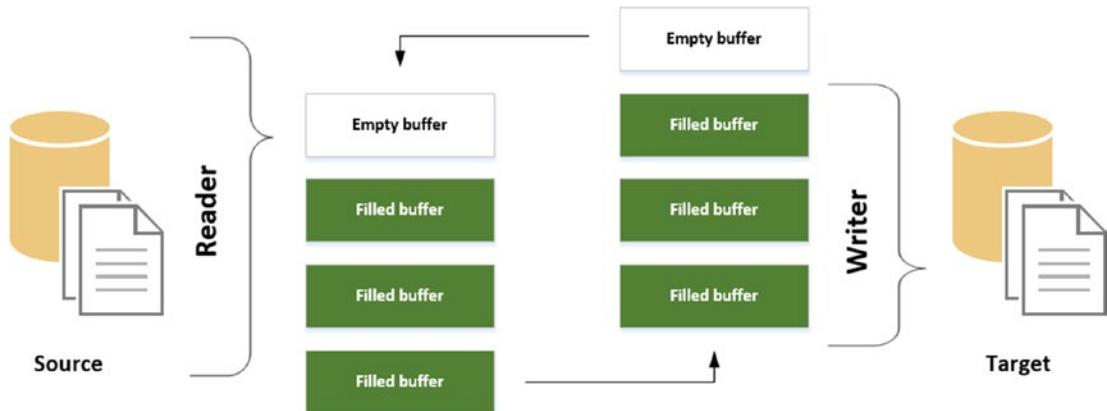


Figure 7-5. Internals of a backup operation

In the previous section where we discussed the BACKUPBUFFER wait type we explained that the BACKUPBUFFER wait type occurs when we are waiting for a free (empty) buffer to become available. For the most part, the BACKUPBUFFER wait type is situated on the left side of Figure 7-5, at the reader. The BACKUPIO wait type occurs for the most part on the right side of Figure 7-5, at the writer section. When BACKUPIO waits occur, there is a delay in the time the writer is writing data. This delay can be caused by many different things; for instance, when writing a backup to a slow disk, writing a backup to a network location, or when restoring a database.

The BACKUPIO wait type will frequently be accompanied by ASYNC_IO_COMPLETION waits when a database backup or restore is performed.

BACKUPIO Example

We can make use of the same example as we used to demonstrate the BACKUPBUFFER wait type. I did modify the query a little to return BACKUPIO waits instead of BACKUPBUFFER waits, and I also included the ASYNC_IO_COMPLETION in the results of the query against the sys.dm_os_wait_stats DMV. Listing 7-4 shows the modified backup query.

Listing 7-4. Generating BACKUPIO waits

```
-- clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

BACKUP DATABASE [AdventureWorks]
TO DISK = N'F:\Backup\aw_21042015.bak'
WITH
NAME = N'AdventureWorks-Full Database Backup';
GO

-- Query BACKUPIO waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'BACKUPIO'
OR wait_type = 'ASYNC_IO_COMPLETION';
```

The results of the query against the `sys.dm_os_wait_stats` DMV can be seen in Figure 7-6.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	ASYNC_IO_COMPLETION	2	1017	1017	0
2	BACKUPIO	159	690	21	1

Figure 7-6. ASYNC_IO_COMPLETION and BACKUPIO waits

As you can see in Figure 7-6, the database backup caused both wait types to be generated, and the most time has been spent on the ASYNC_IO_COMPLETION wait, which is responsible for reading the data pages that need to be written to the backup file. Since my backup destination is on an SSD disk, we didn't encounter very high BACKUPIO wait times.

Lowering BACKUPIO Waits

Tweaking the BUFFERCOUNT and MAXTRANSFERSIZE options do not have as much impact on the BACKUPIO wait type as they did on the BACKUPBUFFER wait type. When you see higher than normal wait times on the BACKUPIO wait type, the problem is most likely

related to the throughput of either your storage subsystem or network location you are writing or reading your backup to/from. Make sure to check both locations for possible performance problems like high latency or network utilization.

BACKUPIO Summary

Just like the BACKUPBUFFER wait type, the BACKUPIO wait type occurs when a backup or restore operation is being performed. While the BACKUPBUFFER wait type is mostly related to the speed at which the backup operation can access the backup buffers, the BACKUPIO wait type is related to the speed at which those backup buffers can be written to disk. BACKUPIO waits frequently occur together with ASYNC_IO_COMPLETION waits when performing full database backups or restores. When seeing higher than normal wait times for the BACKUPIO wait type, check the performance metric of the location you are writing or reading the backup file to or from. Lowering BACKUPIO wait times will not have an impact on the query performance of your system, but will help speed up backup and restore operations.

BACKUPTHREAD

The BACKUPTHREAD wait type is frequently seen when performing restore operations on a database, but can also occur during a backup operation. It occurs when another thread is waiting for the backup/restore operation to finish so it can continue processing.

What Is the BACKUPTHREAD Wait Type?

When you see BACKUPTHREAD waits occurring, it means that another thread wants to access a resource that is currently being accessed by a backup or restore operation. During the time the thread has to wait for the backup/restore to complete, BACKUPTHREAD wait time will be recorded. An example of this type of wait would be a thread that wants to access the database data file while it is being restored; for instance, the ASYNC_IO_COMPLETION wait type that is writing the data file to disk.

BACKUPTHREAD waits are not usually a cause for concern. They only indicate that other threads are waiting for the backup/restore operation to complete, and they frequently have the same duration as the time it took for your backup or restore to complete. They do, however, give you a hint that there are other waits occurring that might deserve investigation if the wait times are higher than expected.

Because a picture says more than a thousand words, Figure 7-7 shows the relation of the BACKUPTHREAD wait type with a restore operation and other waits occurring.

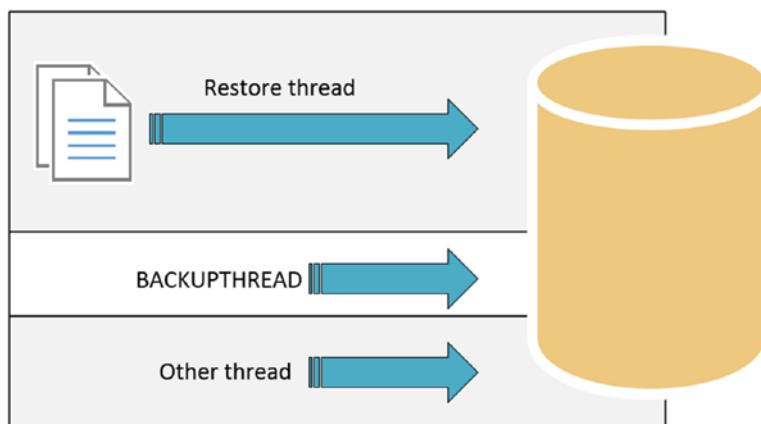


Figure 7-7. BACKUPTHREAD relation to other threads

In Figure 7-7 you can see that the BACKUPTHREAD wait is occurring because another thread also wanted to access a resource that was currently owned by the restore operation.

BACKUPTHREAD Example

An easy way to demonstrate BACKUPTHREAD waits occurring is by performing a restore operation. When you perform a restore, other processes will need to access the database data files to write the information from the backup file to the database data files.

Listing 7-5 shows a script to restore a backup file I made earlier of the AdventureWorks database on my test SQL Server.

Listing 7-5. Restore AdventureWorks database

```
-- Restore database
USE [master]
RESTORE DATABASE [AdventureWorks]
FROM DISK = N'F:\Backup\AWBackup.bak'
WITH FILE = 1, REPLACE;
GO
```

If we were to look at the `sys.dm_os_waiting_tasks` DMV while the backup is running, we would see the waits occurring as shown in Figure 7-8, which shows a selection of waits on my test SQL Server.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms
1	BACKUPIO	186	462	61
2	BACKUPTHREAD	10	1534	1520
3	PREEMPTIVE_OS_WRITEFILEGATHER	2	831	824

Figure 7-8. BACKUPTHREAD and other waits

As you can see in Figure 7-8, the wait time of the BACKUPTHREAD wait type is pretty close to that of the PREEMPTIVE_OS_WRITEFILEGATHER wait type. This wait type is responsible for writing data to the file system, but we will dive deeper into this specific wait type in Chapter 11, “Preemptive Wait Types.”

Lowering BACKUPTHREAD Waits

While the BACKUPTHREAD wait type itself doesn’t indicate any problems, its combination with other wait types can be a reason for some additional research. Basically, every method you can use to speed up your backup or recovery process will have an impact on the BACKUPTHREAD wait time.

Some good pointers to start with are the `BufferCount` and `MaxTransferSize` options that you can specify on the `BACKUP` and `RESTORE` T-SQL commands. We touched upon these settings when we discussed the `BACKUPBUFFER` and `BACKUPIO` wait types. Tweaking these settings can make your backups and restores take less time, resulting in lower BACKUPTHREAD wait times.

Another setting that can dramatically improve backup and restore times is the instant file initialization option that we discussed in Chapter 6, “IO-Related Wait Types,” in the `ASYNC_IO_COMPLETION` section.

BACKUPTHREAD Summary

The BACKUPTHREAD wait time doesn't indicate access to a specific resource, but rather indicates that another process is waiting for a backup or restore operation to complete. It is very common to see this wait type, especially during restore operations. Lowering the duration of backup and restore operations will also be reflected in the wait times of the BACKUPTHREAD wait type. One of the methods you can use to lower BACKUPTHREAD wait times is checking whether instant file initialization is enabled. This setting does not directly impact the BACKUPTHREAD wait type, but it will impact other wait types, which will in turn impact the BACKUPTHREAD wait time.

CHAPTER 8

Lock-Related Wait Types

Locking is a fundamental part of every relational database, or Relational Database Management System (RDBMS). SQL Server is based on the relational database model, and as such uses locking when data is accessed. Even though we frequently relate locking to performance problems, it plays a vital role in making sure your data is reliable during concurrent workloads. The way SQL Server, or any other RDBMS for that matter, takes care of this data reliability is by following the “ACID” properties, which were originally defined by Jim Gray in the 1970s but received their name in 1983 from Andreas Reuter and Theo Härder. These ACID properties are enforced upon single operations, which we know as transactions. The acronym ACID consists of four characteristics that guarantee data reliability inside transactions. The following list describes each of these characteristics:

- Atomicity: The atomicity characteristic requires that transactions are all or nothing. This means that if one part of the transaction fails, the complete transaction fails, and every change done inside the transaction needs to be changed back to the state before the transaction started.
- Consistency: The consistency characteristic requires that data written to the database by the transaction is legal. This means that the data must be stripped of illegal or bad input.
- Isolation: The isolation characteristic requires that every transaction is hidden from other concurrent transactions. From a transaction point of view, this means every transaction is executed serially.
- Durable: The durable characteristic requires that every committed transaction remains committed, even in the event of a power failure or disaster.

As you might have guessed from reading the different ACID properties, locking inside SQL Server is closely related to the Isolation characteristic.

Since this chapter is dedicated to lock-related wait types, we won't go into detail about ACID properties besides Isolation. If you are interested in learning more about the ACID properties and database theory, a good place to start would be the "Principles of Transaction-Oriented Database Recovery" research paper by Andreas Reuter and Theo Harder, which describes the ACID properties in detail.

To get a better understanding of how the Isolation characteristic works, we need to understand transactions. A transaction represents an interaction with the database that can consist of multiple actions and that is separated from other transactions.

To make sure our transactions do not conflict with other concurrent transactions, SQL Server uses locks. These locks make sure no other transaction can modify data that your transaction is processing at the same time. For example, if you make a \$100 withdrawal from your bank account, you do not want another concurrent withdrawal to modify that amount. Other transactions will have to wait with their withdrawals until your transaction is completed. Inside SQL Server, this process works the same way. When you request data from your database, you want to get the data returned you asked for, without running the risk that the data is being modified while you are requesting it.

When you run your transaction it will be protected by a lock SQL Server places on the object you are accessing. If another transaction wants to interact with the same object, a block will occur. When this block occurs, the latter transaction will have to wait until the lock on the object is removed. The transaction can then place its own lock on the object and start its interaction.

There are many options available to us within SQL Server to control the behavior of locking and blocking, and most of them are related to changing the Isolation of certain or all transactions against a database. There is also a lot of information about locking and blocking we can access inside SQL Server, and not the least of these are inside wait statistics. The time a transaction is waiting to access a locked object is recorded as wait time for specific, lock-related wait types (depending on the type of lock the transaction intends to place).

In this chapter we will discuss the various wait types that are related to locking and blocking and how we can lower or even resolve them. This requires some knowledge of how SQL Server uses locks, and for this reason I included a section to familiarize ourselves with locking and blocking before we dive into the lock wait types.

Introduction to Locking and Blocking

As we just discussed, SQL Server uses locks to isolate different concurrent transactions from each other so data is only accessed or modified by one transaction at a time. There are different lock types, or lock modes, SQL Server can use, and there are various object levels SQL Server can place locks on. To make it even more complex, different lock modes are not necessarily compatible with each other, and when two incompatible locks meet, a block occurs.

Lock Modes and Compatibility

To start off, let's get ourselves familiar with the different types of locks, or lock modes, inside SQL Server. The list that follows describes the most common lock modes. There are more lock modes inside SQL Server, but those only occur when you perform very specific actions. A complete list of the different lock modes can be found on the MSDN page that discusses lock modes here: <https://technet.microsoft.com/en-us/library/ms175519.aspx>. SQL Server uses acronyms to indicate which lock mode is being used inside SQL Server. These acronyms are shown in parentheses:

- Shared (S): A Shared lock will be placed on a resource when a query is selecting data from that resource. For instance, a `SELECT * FROM [table]`.
- Update (U): The Update lock mode is used when a query wants to modify a resource. It was introduced to prevent “deadlocks,” a situation where locks are waiting on each other to release in concurrent transactions that want to modify the same resource.
- Exclusive (X): An Exclusive lock is placed when a transaction wants to modify the resource. When an Exclusive lock is in place, no other transactions can modify the resource. For instance, `INSERT`, `UPDATE`, or `DELETE` T-SQL statements will result in Exclusive locks.
- Schema (Sch): Schema locks are used when a table is being modified. An example of this would be adding a column to a table.
- Intent (I): Intent locks are used to indicate that locks are placed at a lower level in the locking hierarchy. We will go into more detail on the lock hierarchy in a bit.

When different locks need to interact with each other, SQL Server performs a lock compatibility check on the different lock modes involved. Not all of the lock modes are compatible with each other, which means that when two different transactions are not able to access the resource at the same time because of incompatible locks, a block will occur. For instance, when a Shared lock is placed to read from a row, and another transaction wants to modify the row by placing an Exclusive lock, the Exclusive lock will have to wait until the Shared lock is removed. Table 8-1 shows the lock mode compatibility for the Shared, Update, and Exclusive lock modes.

Table 8-1. Lock Compatibility

Lock Mode	Shared	Update	Exclusive
Shared	Yes	Yes	No
Update	Yes	No	No
Exclusive	No	No	No

Let's go through an example to illustrate lock compatibility. Say you want to read from a row inside a table by executing a SELECT statement against that table. When you execute your query, SQL Server will check if there is any existing lock already in place on the row you want to access, and if it is compatible with the lock you want to place on the row. Let's assume there isn't a lock in place when you run your query. In this case a Shared lock will be placed on the row, indicating that your query is reading data from that row. Right after you execute your query another transaction is issued by another user that wants to modify data inside the row you are accessing. SQL Server will detect that there already is a Shared lock in place on the row, making the second transaction wait before placing its Exclusive lock, since Shared and Exclusive locks are incompatible. The user who ran the second transaction might experience a delay, since the transaction is waiting for the Shared lock to be removed before its Exclusive lock can be placed. If a third transaction is started that wants to read the same row as your transaction, no lock conflict will occur. Shared locks are compatible with other Shared locks, meaning that the third transaction does not have to wait to place its lock, and it directly receives the results it asked for.

Figure 8-1 shows the example where the dotted line indicates an incompatible lock that has to wait.

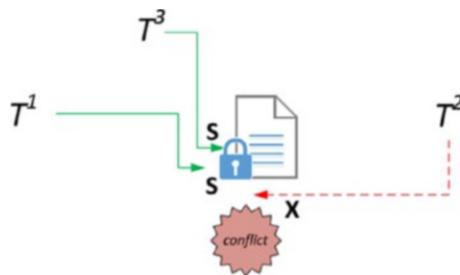


Figure 8-1. Concurrent lock situation

Locking Hierarchy

SQL Server uses multigranular locking to allow different locks for different-level objects. It does this to minimize the overhead cost of locking. The lowest possible object where a lock can be placed is a row, and the largest is the database. There are many levels between those two granularity levels, and SQL Server automatically decides on what level the lock should be placed to minimize locking overhead. The following list shows the most common lock levels, ordered from the highest granularity to the smallest:

- Database
- Database file
- Table/Object
- Extent
- Page
- RID (row inside a heap)/KEY (row inside a clustered index)

The Intent locks we discussed earlier also play an important part in the placement of locks upon the different granularity levels. SQL Server will place Intent locks on objects that are on a higher granularity to indicate a lock has been placed at a lower level. This protects the lower-level locks from changes on objects at a higher granularity level. All the Intent locks that are placed, from the highest granularity level to the actual lock on an object, when looked at together are called the locking hierarchy.

Figure 8-2 shows a graphical representation of a locking hierarchy for the modification of data inside a row, which will require an Exclusive lock on the row and Intent Exclusive locks higher in the hierarchy.

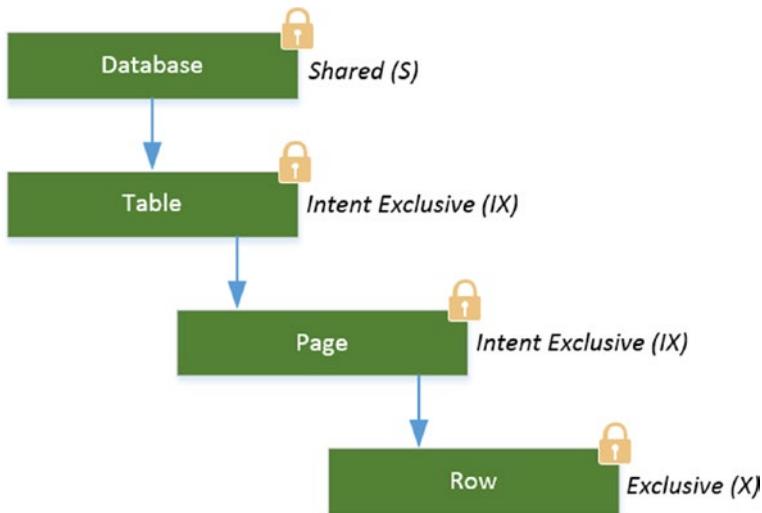


Figure 8-2. Lock hierarchy example

Note the Shared lock on the database level. Every request will always place one to protect changes to the database while transactions are active. This makes sure that, for instance, you cannot delete a database while transactions are still active. Also note that the Intent locks will use the same lock mode on the lowest object, in this case Intent Exclusive (IX). If a Shared lock was placed, the lock mode of the Intent lock would change as well, in this case to Intent Shared (IS). We will go a little deeper into Intent locks a bit further on in this chapter.

Isolation Levels

We can exercise a certain level of control over what locks are being placed by a transaction by changing the Isolation level. The Isolation level defines the degree to which transactions are isolated from each other during concurrent operations. We can change the Isolation level on either a connection or a transaction basis. Changing the Isolation level will only change the behavior of Shared locks; Exclusive locks that are needed for data modification are not affected. Changing the Isolation level will also introduce certain phenomena. These phenomena have an impact on the results of your read transaction and occur because of the changes to how Shared locks are placed and

held during the transaction. The list that follows shows the various Isolation levels, from the lowest form of Isolation to the highest, available in SQL Server and the phenomena related to them:

- Read Uncommitted: This Isolation level will allow reads to occur while another transaction is performing modifications on the same object. It will not wait until the Exclusive lock on the object is released. This makes it possible to read uncommitted values called “dirty reads.” Dirty reads can be bad (if you do not expect them) because they can return a value that is no longer current in the database. For instance, if someone is updating a value to “B” while it was “A” at the start of the transaction, other users that query the same data at the same time can get the old value of “A” back instead of the updated “B” value.
- Read Committed: This is the default Isolation level in SQL Server. Using this Isolation level will make read transactions wait until concurrent write transactions are completed. A Shared lock will be placed on a row and will be released right after the row has been read. The phenomenon associated with this Isolation level is called “inconsistent analysis.” This means that it is possible to receive different results from the same read query if the data were modified by another transaction in the time between both read transactions.
- Repeatable Read: Setting the Isolation level to Repeatable Read will lock rows that are being read by a transaction. But instead of releasing the Shared lock on the row after it has been read, Repeatable Read will keep the lock in place until the entire transaction is completed. A Repeatable Read makes it possible for “phantom reads” to occur. Phantom reads occur whenever data is added or changed by another transaction that has not yet been locked by the read transaction.
- Serializable: The Serializable Isolation level is the highest possible Isolation level you can use, and that means it will place the most locks to ensure the data you are reading is not modified during the time the transaction is running. It does this by locking the entire

range of data (for instance, an entire table) you are selecting, making it impossible to make changes to that data. Since the entire range of data you are selecting is being locked right at the start of the transaction, there are no phenomena possible.

SQL Server 2005 added another method for isolating transactions called Row Versioning. Row Versioning uses versions of data modification and returns them to read queries without causing blocking. When a transaction modifies data, that change will be recorded as a version. When a read transaction accesses the same data, it will receive the version of the change before the modification transaction is committed. More information about Row Versioning can be found on Books Online at <https://technet.microsoft.com/en-us/library/ms189050.aspx>.

Because Isolation levels, and their locking behavior, can be complex to understand, I added Figure 8-3, which shows the way the various Isolation levels implement locking during a read operation. The boxes represent rows inside a table, and a row with a lock means a Shared lock is active on that row.

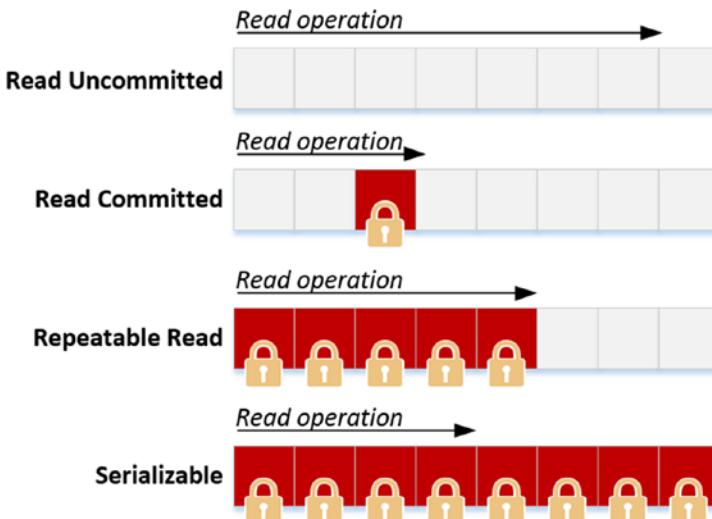


Figure 8-3. Isolation levels and locking behavior

There are various reasons why you would want to use a different Isolation level than the default of Read Committed. In many cases these reasons are related to the amount of locking/blocking you expect with your workload, or how “correct” the data returned by your transaction should be. For instance, with the default Isolation level of Read Committed it is possible that data is modified by other transactions while your transaction is running, which means that the results at the end of the transaction are not the same as they were at the start of your transaction. To make sure no data can change while your transaction is running, you could use the Serializable Isolation level, but this means more locks need to be placed and maintained, resulting in more blocking in concurrent SQL Server environments.

We can only change the default Isolation level of Read Committed by specifically configuring a different Isolation level for a connection or by supplying a table hint (an exception is Snapshot Isolation, which is configured at the database level). For instance, the two queries that follow show two different methods of executing a query using the Read Uncommitted Isolation level. The first query sets the transaction Isolation level for the entire session:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
GO  
  
BEGIN TRANSACTION  
  
SELECT *  
FROM Person.Person  
  
COMMIT TRANSACTION;  
GO
```

Another method is to use a table hint to set the Isolation level to Read Uncommitted:

```
SELECT *  
FROM Person.Person  
WITH (READUNCOMMITTED);
```

Both of these methods will achieve the same effect, but keep in mind that setting the Isolation level for the session will result in using the selected Isolation level for all the queries that are being executed in this specific session after setting it.

Querying Lock Information

To take a look at currently placed locks we can use the `sys.dm_tran_locks` DMV. This DMV will return a row for every active lock inside the SQL Server instance, along with information like the type of lock, the resource type, the session ID that placed the lock, and whether the lock is granted or is waiting to be placed. Figure 8-4 shows a (small) portion of the output of the DMV on my test SQL Server machine.

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_status
45	KEY		5	(36154064afaf)	72057594050052096	0	X	LOCK	GRANT
46	PAGE		5	1:20446	72057594045333504	0	IX	LOCK	GRANT
47	PAGE		5	1:20447	72057594045333504	0	IX	LOCK	GRANT
48	PAGE		5	1:20432	72057594045333504	0	IX	LOCK	GRANT
49	PAGE		5	1:20432	72057594045333504	0	S	LOCK	WAIT
50	KEY		5	(bdeab116bd74)	72057594050052096	0	X	LOCK	GRANT
51	KEY		5	(16336affdf6ae)	72057594050052096	0	X	LOCK	GRANT
52	KEY		5	(8194443284a0)	72057594045333504	0	X	LOCK	GRANT
53	KEY		5	(d8b6f345a521)	72057594045333504	0	X	LOCK	GRANT
54	KEY		5	(b9b173bbe8d5)	72057594045333504	0	X	LOCK	GRANT

Figure 8-4. `sys.dm_tran_locks` output

If we take a look at Figure 8-4 we can see that a number of Exclusive locks (X) have been granted and are placed at the Key lock level. This means a transaction is currently modifying data inside a clustered index. There is also an Intent Exclusive lock on the Page level, which is above the Key lock level, indicating that there is an Exclusive lock lower down in the hierarchy. Also note that a Shared lock is currently waiting to get placed on the same data page (1:20432). The lock cannot be granted just yet, as there is an incompatible Intent Exclusive lock in place.

Since the Shared lock has to wait before it can be placed on the data page, we can view the time it has been waiting by looking at the wait statistics. Figure 8-5 shows a part of the results of a query against the `sys.dm_os_waiting_tasks` DMV.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address
7	0x00000008261B5D088	17	0	110116	SP_SERVER_DIAGNOSTICS_SLEEP	0x0000000000000001	NULL
8	0x00000008261B5DC28	26	0	55911991	HADR_NOTIFICATION_DEQUEUE	0x0000000810155F150	NULL
9	0x0000000825DD004E8	23	0	1335703	BROKER_EVENTHANDLER	NULL	NULL
10	0x0000000825DD01468	9	0	55911597	BROKER_TRANSMITTER	NULL	NULL
11	0x0000000825DD01848	27	0	570	SLEEP_TASK	NULL	NULL
12	0x00000008261B5CCA8	55	0	147659	LCK_M_S	0x00000008252962940	NULL
13	0x00000008266027088	12	0	3115	XE_TIMER_EVENT	NULL	NULL
14	0x00000008266027848	1	0	55912386	WAIT_XTP_HOST_WAIT	NULL	NULL
15	0x00000008266027C28	20	0	48706900	ONDemand_TASK_QUEUE	0x0000000832FC9EB50	NULL

Figure 8-5. Lock information inside `sys.dm_os_waiting_tasks`

By using the `sys.dm_os_waiting_tasks` DMV, we can see that session ID 55 is currently waiting on a resource named `LCK_M_S`. This represents a Shared lock resource type. Session ID 55 is currently being blocked by session ID 53, which happens to be the same session that has the Exclusive and Intent Exclusive locks placed on the objects session ID 55 is trying to query. The `sys.dm_os_waiting_tasks` DMV will also return information we can use as input for the `sys.dm_tran_locks` DMV. This information will be available in the `resource_description` column of the `sys.dm_os_waiting_tasks` DMV, as shown in Figure 8-6.

blocking_session_id	blocking_exec_context_id	resource_description
NULL	NULL	NULL
53	NULL	pagelock fileid=1 pageid=20432 dbid=5 subresource=FULL id=lock8258467e00 mode=IX associatedObjectid=72057594045333504

Figure 8-6. *resource_description column of the sys.dm_os_waiting_tasks DMV during a block*

If we copy the `associatedObjectID` and use it as input in the `WHERE` clause against the `sys.dm_tran_locks` DMV, we will receive more information about why, and on what, this task is waiting. The following query will retrieve all the rows inside the `sys.dm_tran_locks` DMV that have a `resource_associated_entity_id` of 72057594045333504:

```
SELECT *
FROM sys.dm_tran_locks
WHERE resource_associated_entity_id = ' 72057594045333504';
```

On my test SQL Server, the query returned 32 locks, 26 of which are Exclusive locks on rows inside a clustered index; there are also a number of Intent Exclusive locks on data pages and one Shared lock that is waiting to be placed on a page. The waiting Shared lock is the one returned by the `sys.dm_os_waiting_tasks` DMV. A portion of the results is displayed in Figure 8-7.

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_status
1	KEY		5	(e22df2116e)	72057594045333504	0	X	LOCK	GRANT
2	KEY		5	(0932b31cce2)	72057594045333504	0	X	LOCK	GRANT
3	KEY		5	(8bc081279a03)	72057594045333504	0	X	LOCK	GRANT
4	KEY		5	(98ec012aa510)	72057594045333504	0	X	LOCK	GRANT
5	KEY		5	(a1c5936a3c965)	72057594045333504	0	X	LOCK	GRANT
6	KEY		5	(e8a68f387dfa)	72057594045333504	0	X	LOCK	GRANT
7	KEY		5	(d08358b1108)	72057594045333504	0	X	LOCK	GRANT

Figure 8-7. *Lock information from sys.dm_tran_locks*

CHAPTER 8 LOCK-RELATED WAIT TYPES

Finding lock information and figuring out who is blocking whom by querying the sys.dm_tran_locks DMV can be a challenge on systems where you have many locks and blocks occurring, since the DMV will return a row for every lock placed. Another, easier, method to analyze locking and blocking is to use the sp_WhoIsActive stored procedure created by Adam Machanic. This stored procedure will return information about everything that is running at that time and is a great tool for analyzing performance problems. With some extra parameters it will also return a wealth of locking information without you having to join various DMVs yourself. You can download the sp_WhoIsActive DMV from its website at <http://whoisactive.com/downloads/>.

To show you an example of the sp_WhoIsActive stored procedure I ran it on my test SQL Server while it was experiencing a blocking problem. The most basic way of running it is by just executing it:

```
EXEC sp_WhoIsActive;
```

Figure 8-8 shows a small portion of the results. There are many more columns available that will show you additional information, like the session ID of the blocking session.

	dd hh:mm:ss.mss	session_id	sql_text	login_name	wait_info	CPU
1	00 00:12:52.210	53	<?query -- begin transaction update person...	EVDL-SQL2017-01\Administrator	NULL	21,127
2	00 00:10:28.856	55	<?query -- select * from person.Address -?>	EVDL-SQL2017-01\Administrator	(628859ms)LCK_M_S	2

Figure 8-8. sp_WhoIsActive default results

We can directly identify wait statistics information and the queries that are being executed at this time. Because the blocking_session_id column returned a session ID of 55 for the SELECT query, we should take a look at what the query that is being executed by session ID 55 is doing. By clicking the query link inside the sql_text column, we can view the whole query text, as shown in Figure 8-9.

```

<?query --+
begin transaction

update person.Address
set PostalCode = '98019' WHERE PostalCode = '98011'

-- rollback transaction
--?>```

```

Figure 8-9. *sql_text output from sp_WhoIsActive*

In this specific case we can probably resolve the blocking problem pretty quickly. The query that is causing the block has left its transaction open without performing a COMMIT or ROLLBACK. As long as a transaction stays open, locks are being kept in place and are not released.

The `sp_WhoIsActive` stored procedure also has a parameter to return additional locking information, including information about the lock hierarchy. The query that follows will execute the `sp_WhoIsActive` stored procedure and retrieve extra lock information for the queries that are currently executing:

```
EXEC sp_WhoIsActive @get_locks=1;
```

Figure 8-10 returns the new columns that are added to the output of `sp_WhoIsActive`.

	locks	used_memory	status	open_tran_count
1	<Database name="AdventureWorks"><Locks><Lock req...>	3	sleeping	1
2	<Database name="AdventureWorks"><Locks><Lock req...>	4	suspended	0

Figure 8-10. *Lock information returned by sp_WhoIsActive*

By clicking the link below the `locks` column, we can view which locks are being used and on what objects they are placed. This will also give us a good look at the locking hierarchy that is in place for this specific query. Figure 8-11 shows the extra lock information for the first returned query in Figure 8-10.

CHAPTER 8 LOCK-RELATED WAIT TYPES

```
<Database name="AdventureWorks">
  <Locks>
    <Lock request_mode="S" request_status="GRANT" request_count="1" />
  </Locks>
  <Objects>
    <Object name="Address" schema_name="Person">
      <Locks>
        <Lock resource_type="KEY" index_name="IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode" request_mode="X" request_status="GRANT" request_count="1" />
        <Lock resource_type="KEY" index_name="PK_Address_AddressID" request_mode="X" request_status="GRANT" request_count="26" />
        <Lock resource_type="OBJECT" request_mode="IX" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode" request_mode="UIX" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="PK_Address_AddressID" request_mode="IX" request_status="GRANT" request_count="5" />
      </Locks>
    </Object>
  </Objects>
</Database>
```

Figure 8-11. Additional lock information returned by `sp_WhoIsActive`

Here we can see that an Intent Exclusive lock is placed at the OBJECT level, which means a lock has been placed on the object that is displayed in the `Object_name` field of the XML schema—in this case `Address`, which is a table. A level down, at the page level, we also see two Intent Exclusive locks in place inside two different indexes on the `Address` table. And at the bottom level we see the Exclusive locks, which are placed at the KEY objects, which indicate rows inside an index.

There are many more parameters available for the `sp_WhoIsActive` stored procedure, each one of them returning more information about various parts of SQL Server. This makes the `sp_WhoIsActive` Stored Procedure a great tool for finding out what is going on inside your SQL Server instance, and I encourage you to give it a try.

Now that we have discussed many aspects of locking and blocking, from lock modes and hierarchies to analyzing locks and blocks, we should be ready to take a look at the lock-related wait types inside SQL Server. Keep in mind that this introduction to locking and blocking is far from a complete guide to the topic, as going into more detail on how locking and concurrency works inside SQL Server would fill a book by itself.

LCK_M_S

The first lock-related wait type is the `LCK_M_S` wait type. This wait type represents that a task is waiting to place a Shared lock on a resource.

What Is the LCK_M_S Wait Type?

The `LCK_M_S` wait type indicates that a task is, or has, been waiting to place a Shared lock on a resource. It is important to understand that you will only see this wait type when some form of blocking is occurring, since a task is waiting to place the Shared

lock. It doesn't mean there is a Shared lock active on the resource. This is true for every lock-related wait type, as they will only get recorded when there is a blocking situation.

Since the LCK_M_S wait type is related to Shared locks, it will occur when a read action is being performed but has to wait because an incompatible lock is already in place on the resource we want to read. The time we are waiting before we are able to place the Shared lock is recorded as the wait time of the LCK_M_S wait type.

Figure 8-12 shows a common situation that will result in LCK_M_S waits occurring. In this case an Exclusive lock has been placed on a page by T1, indicating a data modification. When T2 wants to read the data from the page, it will need to place a Shared lock, but since Exclusive and Shared locks are incompatible, a LCK_M_S wait occurs.



Figure 8-12. LCK_M_S wait occurring

LCK_M_S Example

Creating an example of a LCK_M_S wait occurring is not very difficult, as we just need to create a block situation between a data modification query and a data read query.

For this example we are going to run the query seen in Listing 8-1 against the AdventureWorks database. This query will begin a transaction and modify a few rows, but it will not commit or rollback the transaction. Since we explicitly indicated this transaction by supplying a BEGIN TRAN, SQL Server will keep the locks in place until we explicitly execute a COMMIT or ROLLBACK command.

Listing 8-1. Start a modification transaction

```
BEGIN TRAN

UPDATE Sales.SalesOrderDetail
SET CarrierTrackingNumber = '4E0A-4F89-AD'
WHERE SalesOrderID = '43661';
```

When we execute the query, we receive a result very quickly; in my case 15 rows were updated. But like I said before, the transaction is not yet finished, so it will remain running, leaving locks on the objects it modified.

So far we aren't causing any blocking, since this is the test SQL Server and no other queries are running. Let's change that and create a blocking situation.

For this we are going to open a second window in SQL Server Management Studio and execute the query seen in Listing 8-2. This will just perform a SELECT against the Sales.SalesOrderDetail table, the same table in which we are currently modifying data.

Listing 8-2. Select data from a table where a modification is being performed

```
SELECT *
FROM AdventureWorks.Sales.SalesOrderDetail;
```

As soon as we run this SELECT query, we notice that no results are returned and that the query will keep running. This is a typical example of a blocking operation where a transaction is modifying data we want to read inside another transaction.

If we were to query the sys.dm_os_waiting_tasks DMV, we would be able to see the LCK_M_S wait type, as shown in Figure 8-13.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1	0x0000008261B63C28	54	0	24072	LCK_M_S	0x000000823BF729C0	NULL	51

Figure 8-13. LCK_M_S wait occurring

The only way the LCK_M_S wait will be resolved is if the incompatible lock is removed. In this case we performed a rollback of the modification transaction we started in the first SQL Server Management Studio window. We do this by running the ROLLBACK command in the same session window. Immediately after performing the transaction rollback we receive the results the SELECT query asked for. Querying the sys.dm_os_waiting_tasks also showed that the LCK_M_S wait was resolved.

Lowering LCK_M_S Waits

Seeing LCK_M_S waits occur does not necessarily have to mean something is wrong. It does, however, indicate that blocking is occurring. If you notice high wait times on the LCK_M_S wait type, it means that someone's read transaction is currently taking a

long time to complete because it has to wait to place the Shared lock. So the first step will be to identify the query that is causing the block. We can do this by using the `sys.dm_os_waiting_tasks` DMV and looking at the `blocking_session_id` column. This is relatively quick to do when there is only a single block active, but can get complex when many concurrent queries are being blocked by other transactions. In this case we have to follow the blocking chain until we find the head blocker (which is the first lock on an object). Another option is to use the `sp_WhoIsActive` stored procedure we discussed in the “Locking and Blocking Introduction” at the start of this chapter. This stored procedure will move through the blocking chain for you, directly displaying the head blocker.

After we have found the query that is causing the blocking to occur, we need to analyze it and see if we can optimize that query. Maybe it is requesting more locks than it actually needs and thus requires a long time to complete. One way to optimize that query would be to look at whether any indexes should be added so fewer rows are required to be locked. Or maybe you could cut the single transaction into multiple transactions that each access fewer objects. Another possible issue that can cause more locking than necessary is out-of-date statistics. Statistics are used as input for a query plan, and if they do not accurately reflect the contents of the table or index, they can lead to a bad query plan, which in turn can lead to more locks than necessary.

Another option would be to change to Isolation level of the read transactions so no Shared locks are needed in order to read the data. For instance, setting the Isolation level to Read Uncommitted will not place Shared locks, and the read transaction will not be blocked. This does introduce another problem related to the Isolation level, dirty reads, which we discussed in the “Locking and Blocking Introduction” section of this chapter. Next to using Read Uncommitted, you could also use Snapshot Isolation, which will result in fewer Shared locks, but will not cause dirty reads. Snapshot Isolation does put more load on the TempDB database, since it must maintain versions of data if many concurrent transactions are modifying that data.

LCK_M_S Summary

The LCK_M_S wait type occurs when an incompatible lock is being placed on a resource and another transaction wants to place a Shared lock on the same resource. Seeing the LCK_M_S wait type means transactions are being blocked. You should try to identify which queries are causing the block to occur and see if these can be optimized to result

in fewer locks or locks that have a shorter duration. As a final resort you could choose to change the Isolation level of your read transactions, though this does introduce other side effects, like dirty reads or increased load inside TempDB.

LCK_M_U

LCK_M_U wait types are related to locks that use the Update (U) mode. When a task wants to place an Update lock on a resource but an incompatible lock is already in place, LCK_M_U waits occur.

What Is the LCK_M_U Wait Type?

The Update lock type is a special type of lock mode that indicates that data modification is about to occur. Even though its name might suggest it is only related to UPDATE queries, Update locks can also appear when performing INSERT or DELETE statements.

Update locks primarily exist to prevent deadlocks from occurring. Deadlocks indicate that two transactions that want to modify the same object are waiting indefinitely on each other to acquire an Exclusive lock on the resource. To understand how a deadlock situation can occur, and how Update locks can prevent this, take a look at the following scenario that would occur when no Update locks are used.

When two concurrent transactions want to perform a modification on the same object, both transactions would first place a Shared lock on the resource while the data they intended to modify was located. Since Shared locks are compatible with other Shared locks, both transactions would not block each other. When one of the two transactions found the data it needed to modify, it would convert its Shared lock to an Exclusive lock, and then a problem would occur. Since Shared locks are incompatible with Exclusive locks, and since the other transaction would also have a Shared lock on the resource, the conversion from Shared lock to Exclusive lock would not occur. The transaction would need to wait until the Shared lock of the other transaction was removed before it could convert its own Shared lock to an Exclusive lock, but since the other transaction also wants to convert its Shared lock to an Exclusive lock, both transactions would end up waiting on each other, and a deadlock would occur. SQL Server will automatically detect deadlock situations and choose one of the deadlocked transactions as a victim and perform a rollback of that transaction, ending the deadlock situation. Figure 8-14 shows a graphical representation of that situation.

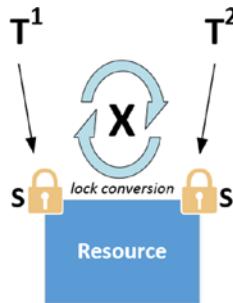


Figure 8-14. Deadlock during lock conversion

When Update locks are used inside SQL Server, no deadlock situation could occur. Update locks are compatible with Shared locks, but not with Exclusive or other Update locks. In the preceding scenario, the first transaction to find the data it needed to modify would not directly convert to an Exclusive lock, but rather would convert to an Update lock first. Since Update and Shared locks are compatible, there would be no problem converting to an Update lock, even though there was a Shared lock in place from the other transaction. The Update lock would then get converted to an Exclusive lock so the data modification could occur. Figure 8-15 shows this lock behavior.

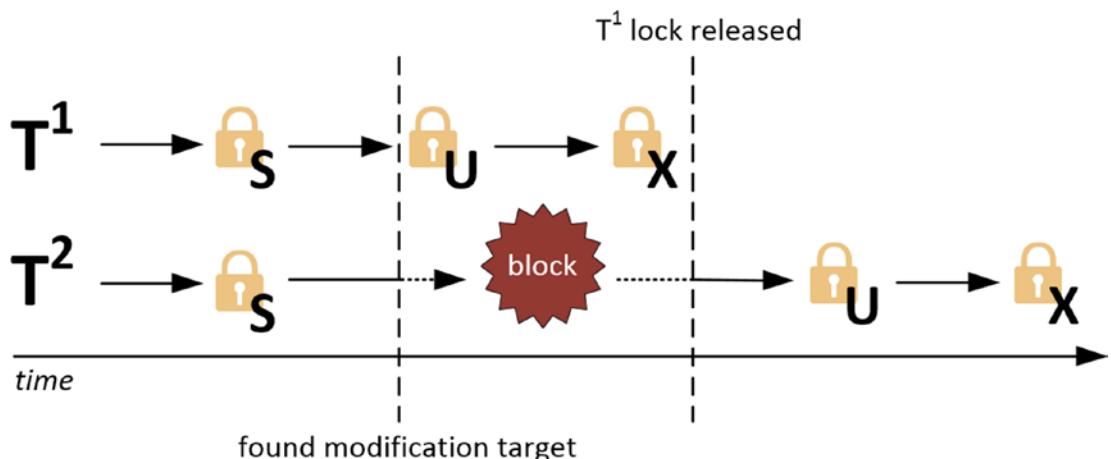


Figure 8-15. Update locks during concurrent data modifications

When a transaction wants to place an Update lock but there is an incompatible lock already in place on the object, for instance, an Exclusive lock, the LCK_M_U wait type will be recorded.

LCK_M_U Example

To show you an example of LCK_M_U waits occurring, we have to create a situation where concurrent transactions want to modify the same resource. For this we are going to make use of the Ostress utility to execute an identical query using multiple connections. The query I am going to execute can be seen in Listing 8-3. This will perform an UPDATE against the Person.Address table inside the AdventureWorks database. I saved the query inside a .sql file named LCK_M_U.sql.

Listing 8-3. Modify the Person.Address table

```
UPDATE Person.Address
SET City = 'Los Angeles'
WHERE StateProvinceID = 9;
```

After saving the file I run the Ostress utility using the following command:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks2012 -i"C:\lck_m_u.sql" -n150 -r5 -q
```

This will create 150 concurrent connections, each one executing the query in Listing 8-3 five times. This should be enough to create some blocking.

While the Ostress utility is running, I query the sys.dm_os_waiting_tasks DMV to find out what tasks are waiting. A small portion of the results are shown in Figure 8-16.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1	0x0000008259FB0108	179	0	3491	LCK_M_U	0x0000008246067940	0x0000008254762108	161
2	0x000000825BC87C28	181	0	3486	LCK_M_U	0x000000823BF6E540	0x0000008254762108	161
3	0x000000825BC87468	182	0	3485	LCK_M_U	0x000000823D03840	0x0000008254762108	161
4	0x0000008259FB04E8	183	0	3485	LCK_M_U	0x000000825830B280	0x0000008254762108	161
5	0x0000008259FB08C8	185	0	3480	LCK_M_U	0x0000008246F73B40	0x0000008254762108	161
6	0x0000008259FB0CA8	187	0	3468	LCK_M_U	0x000000825623BA80	0x0000008254762108	161
7	0x0000008259FB1088	190	0	3454	LCK_M_U	0x0000008246C55740	0x0000008254762108	161
8	0x0000008259FB1848	191	0	3453	LCK_M_U	0x000000823E281EC0	0x0000008254762108	161
9	0x0000008259FB1C28	194	0	3442	LCK_M_U	0x0000008256267D40	0x0000008254762108	161

Figure 8-16. LCK_M_U waits occurring

As you can see in Figure 8-16, many different sessions are waiting to acquire an Update lock but are all being blocked by session ID 161. If we query the sys.dm_tran_locks DMV for lock information about this session, we can see it is granted an incompatible Exclusive lock, as shown in Figure 8-17.

	request_mode	request_type	request_status	request_reference_count	request_lifetime	request_session_id
1	X	LOCK	GRANT	0	3355453	161
2	IX	LOCK	GRANT	0	3355453	78
3	S	LOCK	GRANT	1	0	58

Figure 8-17. Session ID 161 holding an Exclusive lock

All the other sessions will have to wait until the Exclusive lock of session ID 161 is removed. Then one of those sessions will acquire the Update lock it is requesting, convert it into an Exclusive lock, and perform its modification. That cycle will repeat until all the sessions are done with their modifications.

Lowering LCK_M_U Waits

Lowering LCK_M_U waits uses the same approach as lowering LCK_M_S wait types: try to identify the transaction that is causing the blocking to occur and try to optimize its locking behavior.

Changing the Isolation level will have little effect on LCK_M_U wait times since other Isolation levels have the most impact on transactions that perform reads. This makes optimizing your queries and/or indexes the way to go if you need to lower higher-than-normal wait times on the LCK_M_U wait type.

LCK_M_U Summary

The LCK_M_U wait type is related to locks that use the Update lock mode. Update locks are used to prevent deadlocks from occurring when concurrent transactions try to convert their Shared locks to Exclusive locks. Lowering LCK_M_U wait times is primarily achieved by optimizing potential blocking queries or indexes.

LCK_M_X

Another of the most common lock-related wait types is the LCK_M_X wait type. Just like both lock-related wait types we have already discussed, the LCK_M_X wait type is related to a specific lock type, in this case the Exclusive lock. And just like the other two lock-related wait types, seeing this wait type means there is some form of blocking occurring.

What Is the LCK_M_X Wait Type?

The LCK_M_X wait type occurs when a task is waiting to place an Exclusive lock on an object. Since Exclusive locks are not compatible with just about any other lock mode, including other Exclusive locks, seeing blocking occur when there are many concurrent modifications is pretty common. This means that seeing LCK_M_X waits occur is pretty common as well, especially in systems that have a high amount of concurrent transactions.

LCK_M_X Example

To demonstrate LCK_M_X waits occurring we are going to execute a SELECT statement without committing it. Before we run the SELECT, we are going to set the Isolation level to Repeatable Read. Doing so makes sure the Shared locks are not removed while the transaction is still running. Since we do not end the transaction, the locks will remain on the objects until we either kill the transaction or perform a COMMIT or ROLLBACK.

The query that follows shows the SELECT statement we will execute against the AdventureWorks database:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ  
BEGIN TRANSACTION  
SELECT *  
FROM HumanResources.Employee;  
-- COMMIT
```

Notice we commented out the COMMIT section to make sure the locks remain in place. Executing the query returns results pretty quickly, after just 1 second I got all the rows of the HumanResources.Employee returned. If we query the sys.dm_tran_locks DMV, we should see that all the Shared locks are still in place, as shown in Figure 8-18.

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type
13	KEY		5	(ab3b33cc1933)	72057594046644224	0	S	LOCK
14	KEY		5	(6835335383db)	72057594046644224	0	S	LOCK
15	KEY		5	(31178495a25a)	72057594046644224	0	S	LOCK
16	KEY		5	(f219840a38b2)	72057594046644224	0	S	LOCK
17	KEY		5	(9517126d379e)	72057594046644224	0	S	LOCK
18	KEY		5	(561912f2ad74)	72057594046644224	0	S	LOCK
19	KEY		5	(03ba5348cf5)	72057594046644224	0	S	LOCK
20	KEY		5	(c7d268878d3a)	72057594046644224	0	S	LOCK
21	KEY		5	(cc35a5ab161d)	72057594046644224	0	S	LOCK

Figure 8-18. Shared locks still in place

While the locks are still in place, we will run another query inside a new window in SQL Server Management Studio. The query that follows will perform an UPDATE on a single row inside the same HumanResources.Employee table:

```
UPDATE HumanResources.Employee
SET JobTitle = 'Tester'
WHERE BusinessEntityID = 5;
```

As soon as we execute the preceding query, we'll notice a block occurring since the query keeps running without returning any results. This is as expected since there is a Shared lock in place on the row, or specifically the index key, that prevents us from updating it.

When we look at the sys.dm_os_waiting_tasks DMV, shown in Figure 8-19, we will notice that the query in the second window is waiting to place an Exclusive lock, indicated by the LCK_M_X wait type.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1	0x00000008252F34CA8	52	0	21826	LCK_M_X	0x0000000824741B400	NULL	55

Figure 8-19. LCK_M_X wait occurring

If we end the SELECT query, by either executing the COMMIT statement or by closing the windows inside SQL Server Management Studio, the Shared locks are removed and the second query will be able to execute its UPDATE command, ending the LCK_M_X wait.

Lowering LCK_M_X Waits

To lower LCK_M_X wait times, you should use the same approach as for lowering other lock-related wait types. Try and identify what queries are causing the blocking and see if you can optimize them so they cause less blocking.

LCK_M_X Summary

The LCK_M_X wait type is related to Exclusive locks being blocked by other locks already in place on the same resource. Since Exclusive locks are incompatible with just about every other lock type, seeing LCK_M_X waits occurring is not uncommon for SQL Server instances that experience concurrent query execution.

LCK_M_I[xx]

Seeing the LCK_M_I[xx] wait type means that a task is being blocked when placing an Intent lock. Since we already discussed the various lock modes on objects, I replaced the lock mode used for the Intent lock as [xx] when discussing this wait type. The [xx] can be replaced by a variety of different lock modes; for instance, a block on an Intent Shared lock would be represented by the LCK_M_IS wait type, while a block on an Intent Exclusive lock would be shown as LCK_M_IX.

What Is the LCK_M_I[xx] Wait Type?

LCK_M_I[xx] wait types indicate that a task is waiting to place an Intent lock on an object. As we learned from the “Introduction to Locking and Blocking” section at the start of this chapter, Intent locks indicate that a lock of the same type is placed on an object lower down in the locking hierarchy. This doesn’t mean Intent locks are only there to warn SQL Server that there is a lock further down the hierarchy. Intent locks behave just like any other lock, and it is entirely possible that one Intent lock can block another, incompatible, Intent lock. Intent locks do have a little bit more flexibility regarding other incompatible Intent locks. For instance, it is possible for two Intent Exclusive locks to exist on the same page object, indicating that a row is going to be modified. It is even possible to have an Intent Shared lock on a page object together with an Intent Exclusive, because both of the locks can read and/or modify different rows.

Next to indicating the type of lock that exists lower down in the locking hierarchy, Intent locks have a few “special” modes the other lock modes do not have. It is possible for Intent locks to represent more than one lock mode on lower levels of the locking hierarchy. The list that follows describes these three Intent lock modes:

- Shared with Intent Exclusive (SIX): This lock mode represents that there are Shared Locks on all objects at a lower level, and Intent Exclusive locks on some of these objects. These locks are acquired by one transaction that wants to read data and plans to modify other data at the same time. When a task is being blocked while trying to place the SIX lock, it will be recorded by the LCK_M_SIX wait type.
- Shared Intent Update (SIU): This lock mode is a combination of Shared and Intent Update locks. Again, it is possible for a single transaction to acquire, and hold, both these lock modes at the same time at a lower level. If a block occurs while trying to place this lock, the LCK_M_SIU wait type will be used to record the wait time.
- Update Intent Exclusive (UIX): This lock mode is another combination of two other lock modes, Update and Intent Exclusive. Blocks on this lock mode will be represented by the LCK_M_UIX wait type.

Seeing high wait times on Intent locks is not very common, since Intent locks are a lot more flexible regarding their incompatibility with each other. This means there generally is less blocking on the Intent level than there is further down the locking hierarchy.

LCK_M_I[xx] Example

In this example we will generate a wait of the LCK_M_IX wait type. This means a transaction is waiting to acquire an Intent Exclusive lock on a higher level in the locking hierarchy.

We will use more or less the same example as we did for the LCK_M_X wait type by running a SELECT statement using the REPEATABLE READ Isolation level and not completing the transaction. The query that follows is the query I will be running against the AdventureWorks database, Person.Address table:

CHAPTER 8 LOCK-RELATED WAIT TYPES

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

```
BEGIN TRAN
```

```
SELECT * FROM Person.Address;
```

```
--COMMIT
```

The COMMIT command has been commented out to leave the transaction open.

If we take a look at the `sys.dm_tran_locks` DMV, shown by Figure 8-20, we see that while the query is running there is only one lock currently active, a Shared lock on the OBJECT resource type. This indicates that the entire table is locked. Since this lock exists on this high level, there is no need for other Shared locks further down the hierarchy.

resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_stal
1 DATABASE		5		0	0	S	LOCK	GRANT
2 DATABASE		5		0	0	S	LOCK	GRANT
3 OBJECT		5		373576369	0	S	LOCK	GRANT

Figure 8-20. Shared lock on a table

If another transaction wants to update a row inside the same table, it would first try to acquire an Intent Exclusive lock on the table and page level before it could acquire an Exclusive lock on the row level. The query that follows is such a transaction, and in this case we will try to update a single row:

```
UPDATE Person.Address  
SET AddressLine1 = '1227 Shoe St.'  
WHERE AddressID = 5;
```

You'll notice that the preceding query "hangs" as long as the SELECT query still has its Shared lock on the table. Even though Intent locks are in most cases compatible with other Intent locks on the same object, having, in this case, a Shared lock on the table level while trying to perform a data modification lower in the hierarchy will cause a block to occur. Shared locks and Intent Exclusive locks are not compatible.

If we look at the `sys.dm_os_waiting_tasks` DMV, we should be able to see that the task to place the Intent Exclusive lock is waiting, as shown in Figure 8-21.

waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1 0x0000008252F34CA8	54	0	15031	LCK_M_IX	0x000000821CEF9640	NULL	52

Figure 8-21. LCK_M_IX wait occurring

Lowering LCK_M_I[xx] Waits

Just like with the other lock-related wait types we discussed earlier, try to focus on the queries that are causing the blocking when trying to lower LCK_M_I[xx] wait times. Because LCK_M_I[xx] waits only occur when incompatible locks are being held on objects higher in the locking hierarchy, it can be worth the time to investigate why those locks are placed so high in the hierarchy. Lock escalation can cause this to happen. Lock escalation occurs when it is more efficient for SQL Server to place a single lock higher in the locking hierarchy instead of locking many objects lower down. For instance, instead of placing thousands of Shared locks on rows, SQL Server can decide to place a single Shared lock on the table level. This requires far less resources to place and maintain than thousands of single locks. As a matter of fact, this is exactly what is occurring in the example I have shown you of the LCK_M_IX wait type. The Person.Address table we are querying with the SELECT query has more than 19,000 rows inside it. When we ran our SELECT * query against the table, it would mean that at least 19,000 row locks would be needed. Because placing and holding that many locks would take a great deal of resources, SQL Server decided to place a single Shared lock on the table instead of 19,000 locks on the rows.

If we can rewrite the query so it requires fewer locks, for instance, by only selecting the first x rows instead of everything, SQL Server would probably choose to lock the rows again, instead of the entire table.

LCK_M_I[xx] Summary

The LCK_M_I[xx] wait type is related to Intent locks, or rather, cases when another incompatible lock is blocking the placement of an Intent lock. Intent locks are placed on higher-level objects to indicate that a lock has been placed on a lower level in the locking hierarchy. Unlike the lock modes we discussed earlier that only represent one type of lock, Intent locks can represent different lock modes lower down in the locking hierarchy. One common cause of high wait times on LCK_M_I[xx] wait types is cases when SQL Server escalates lower-level locks to a higher-level lock. In this situation Intent locks will be blocked and cannot be acquired.

LCK_M_SCH_S and LCK_M_SCH_M

The last two lock-related wait types I want to discuss in this chapter are the LCK_M_SCH_S and the LCK_M_SCH_M wait types. Both of these wait types are related to locks that are being placed on tables, the so-called Schema locks. We didn't give a lot of attention to Schema locks earlier in this chapter, but since they can have a pretty big impact on wait times when they occur, I wanted to include them.

What Are the LCK_M_SCH_S and LCK_M_SCH_M Wait Types?

The LCK_M_SCH_S and LCK_M_SCH_M wait types are both related to Schema locks. Schema locks are placed at the table level to protect the table from modifications while queries access the table, or to prevent queries from accessing the table while it is being modified. There are two different types of Schema lock, Schema Stability (Sch-S) and Schema Modification (Sch-M). Each of them has a different wait type associated with them when a task is being blocked from placing a Schema Stability or Schema Modification lock. The LCK_M_SCH_S wait type (to indicate read access to the table) is recorded when a Schema Stability lock has to wait before it can get placed, and the LCK_M_SCH_M wait type (to indicate the table schema will be changed) is recorded when a Schema Modification lock is waiting to get placed.

Both Schema locks have pretty extreme compatibility with other lock types. The Schema Stability lock is compatible with all other types of locks except for the Schema Modification lock. The Schema Modification lock, on the other hand, is incompatible with every other lock type, including Intent locks.

When using Schema Stability locks it is impossible to modify or change the table in any way while queries are currently reading or writing from or to that table. Because Schema Stability locks are compatible with every lock mode (except for Schema Modification), it is completely normal to see a Schema Stability lock on the table level together with, for example, an Intent Exclusive lock to indicate data modification is occurring on a lower level inside the table.

Schema Modification locks are the opposite from Schema Stability locks, as they prevent any queries from accessing a table while a modification to the table is being performed.

LCK_M_SCH_S and LCK_M_SCH_M Example

For the first example, I am going to add a new column to an existing table, and just as we did in the examples earlier in this chapter, I am going to keep the transaction open by not supplying a COMMIT or ROLLBACK command. The query that follows adds an extra column to the Person.Address table in the AdventureWorks database, but I left the ROLLBACK command commented so the locks stay in place:

```
BEGIN TRAN

ALTER TABLE Person.Address
    ADD
        Test VARCHAR(10);

--ROLLBACK
```

In a new window in SQL Server Management Studio, I am going to execute a simple SELECT query against the Person.Address table, like the one here:

```
SELECT *
FROM Person.Address;
```

If we take a look at the sys.dm_tran_locks DMV while both queries are running, we should be able to see if there is any blocking going on. Figure 8-22 shows a part of the output of a SELECT * query against the sys.dm_tran_locks DMV.

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_stal
1	OBJECT	5		373576369	0	Sch-M	LOCK	GRANT	
2	OBJECT	5		373576369	0	Sch-S	LOCK	WAIT	

Figure 8-22. Sch-M and Sch-S locks

As you can see from Figure 8-22, the first query we started, with the goal of adding a column to the Person.Address table, resulted in a Sch-M lock on the table. The second SELECT query is waiting to receive a Sch-S lock on the same table.

If we query the sys.dm_os_waiting_tasks DMV, we should see a task that is waiting on the LCK_M_SCH_S wait type. Figure 8-23 shows the output of sys.dm_os_waiting_tasks while both queries are running.

CHAPTER 8 LOCK-RELATED WAIT TYPES

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1	0x000000822C572CA8	54	0	103012	LCK_M_SCH_S	0x0000008237B8E980	NULL	52

Figure 8-23. LCK_M_SCH_S wait occurring

Just as we expected, the SELECT query is waiting to acquire its Schema Stability lock.

If we were to reverse the example by starting a read transaction and leaving it open, and then try to modify the same table, we should run into a LCK_M_SCH_M wait, since we can only acquire a Schema Modification lock when there are no active transactions inside the table we want to modify.

To show this situation I executed the query that follows. This starts a SELECT query with the Repeatable Read Isolation level, but I am leaving the transaction open so the locks stay in place:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT * FROM
Person.Address;
-- COMMIT
```

In a new window inside SQL Server Management Studio, I am going to execute the table modification query we used earlier to demonstrate the LCK_M_SCH_S wait type, but without leaving the transaction open:

```
ALTER TABLE Person.Address
ADD
Test VARCHAR(10);
```

As you will probably notice when executing the second query, nothing is returned and the query keeps running, a clear indication of a block occurring.

Let's take a look at the sys.dm_tran_locks DMV again to see what we can find out. Figure 8-24 shows the output on my test SQL Server.

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_st
1	OBJECT		5		373576369	0	S	LOCK	GRANT
2	OBJECT		5		373576369	0	Sch-M	LOCK	WAIT

Figure 8-24. Sch-M lock waiting to be acquired

In this case the table has a Shared lock on it from the SELECT query. Because we are selecting information from a pretty large table, SQL Server decided to place a table lock instead of placing locks on a lower level. Because a Schema Modification lock is incompatible with every other lock type, a block occurs, and we will have to wait until the Shared lock is gone before we can perform our table modification.

Looking at the sys.dm_os_waiting_tasks DMV shows us the results we are expecting, a LCK_M_SCH_M wait, as shown in Figure 8-25.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address	blocking_task_address	blocking_session_id
1	0x00000082595EBC28	54	0	77221	LCK_M_SCH_M	0x0000008237B8F680	NULL	52

Figure 8-25. LCK_M_SCH_M wait occurring

Lowering LCK_M_SCH_S and LCK_M_SCH_M Waits

When you see waits occurring of either the LCK_M_SCH_S or LCK_M_SCH_M wait type, there is probably a transaction active that wants to modify the table. In the case of high wait times on the LCK_M_SCH_S wait type, the table modification transaction is already running; when seeing LCK_M_SCH_M waits, the modification is waiting for all active transactions to remove their locks on the table.

Modifying a table is not something that happens every day on production SQL Server instances (hopefully). Changing large tables can especially be problematic and a cause for high LCK_M_SCH_S wait times, and users that are trying to query the table that is being modified will notice delays. If, however, you absolutely need to modify a table, but there are some long-running queries retrieving information from that table, you can expect LCK_M_SCH_M waits.

Lowering the wait times of both wait types is directly related to performing modifications to tables. A suggestion could be to perform the table modification after office hours, or when there are as few as possible concurrent transactions accessing the table, instead of doing the modification when there are many transactions active against the table.

LCK_M_SCH_S and LCK_M_SCH_M Summary

The LCK_M_SCH_S and LCK_M_SCH_M wait types are the result of Schema Stability or Schema Modification locks being blocked by other locks. Seeing high wait times of either wait type indicates that either a table modification is waiting for all active locks on that table to be removed, or a table modification is currently running and other transactions are being blocked by it.

CHAPTER 9

Latch-Related Wait Types

In Chapter 8, “Lock-Related Wait Types,” we took a pretty deep look at locking and blocking inside SQL Server, together with different wait types that indicate blocking is occurring. Latches look a lot like the locks we discussed earlier; in some cases they even appear to use the same modes as the locks we discussed in Chapter 8. Make no mistake though, latches are completely different than locks, even though they seem to share some features. While locks are used to guarantee transactions are isolated and consistent, latches are used to guarantee the consistency of in-memory objects.

Latches are, just like locking and blocking, a pretty complex subject inside SQL Server. Latches even have their own latch-statistics DMV that records how much time has been spent waiting on specific latch types.

Because of the complexity of latches and their function inside SQL Server, I believe they require an introduction to better understand how you can troubleshoot latch-related wait types later in this chapter. For this reason, we will start this chapter with an introduction to latches, just as we did with the introduction to locking inside Chapter 8, “Lock-Related Wait Types.”

Introduction to Latches

Microsoft describes latches as “lightweight synchronization objects that are used by various SQL Server components” on Books Online. This description is pretty vague, and there is a lot more depth to latches than the description would initially suggest.

The first thing that is important to understand about latches, which we lightly touched upon in the introduction of this chapter, is that latches are completely different than locks. I have heard and read various discussions about latches that treat latches as if they were locks. This confusion is easily explained, as latches, at first glance, do look similar to locks as regards their behavior and naming conventions within SQL Server. Just like locks, latches have various “modes,” and some of the acronyms to indicate

the type of mode used are the same as for some lock modes. Another thing locks and latches have in common is that both objects play a role in keeping SQL Server objects consistent, and the way they manage this seems identical. While locks are used to make sure transactions are consistent, protecting the transaction for the entire duration it is running, latches are only used for the duration they are necessary and are not bound to the duration of a transaction. During the duration of one transaction, many different latches will be acquired and released again. Figure 9-1 visualizes this behavior.

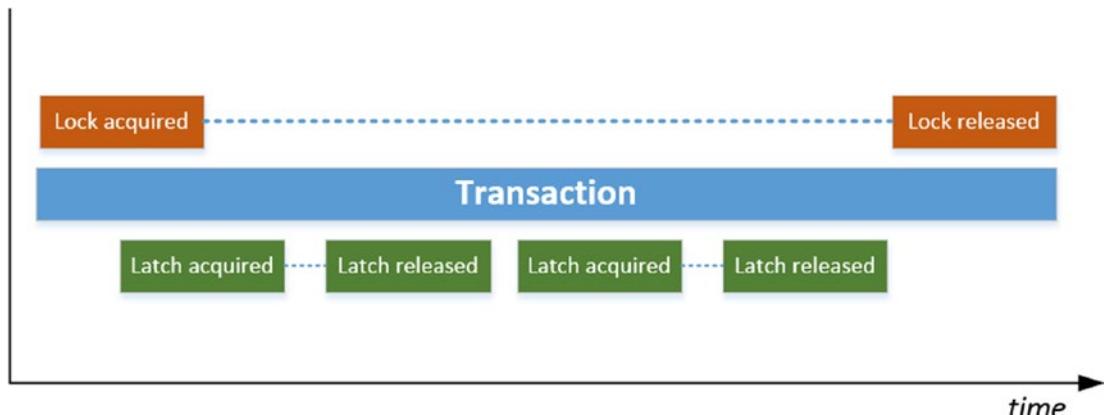


Figure 9-1. Lock and latch behavior during transactions

Placing and especially maintaining locks on SQL Server objects is an expensive operation, mostly because they need to stay in place during entire transactions. Because latches are only needed for specific operations, and are then released again, they are far less costly to use than locks. This explains the “lightweight” part in the latch definition Microsoft uses.

The second part of the latch definition, “synchronization object,” we discussed earlier in this book, but not under that exact name. If you have read through this book so far, you should have noticed, especially in Chapter 6, “IO-Related Wait Types,” that SQL Server uses various methods to handle concurrent threads accessing objects. In Chapter 6, “IO-Related Wait Types,” we talked about mutual exclusion, which makes sure only one thread at a time can access a memory object. In the same chapter we also discussed semaphores that implement gates to limit concurrent access to memory. Latches are another method used to make sure concurrent threads do not threaten the consistency of in-memory objects, and it does this in a way that looks a lot like locking.

Latch Modes

Latches have five different modes available to use when accessing objects. The list that follows describes these five modes, some of which might look familiar:

- SH: The SH mode represents a Shared latch. This mode is used when the latch is reading page data.
- UP: The UP latch mode is used by Update latches that are used whenever a page needs to be modified. By using the Update latch the page can still be read by other latches.
- EX: The EX latch mode, or Exclusive latch, is also used when page modification occurs. Unlike the Update latch, the Exclusive latch does not allow read or write access by other latch modes.
- KP: The KP latch mode is used by Keep latches. Keep latches are used to protect the page so it cannot be destroyed by the Destroy latch. They are compatible with every other latch mode except for the Destroy mode.
- DT: The DT latch mode indicates Destroy latches. Destroy latches are used when removing contents from memory; for instance, when SQL Server wants to free up a data page in memory.

As you can see in this list, the first three latch modes look a lot like those used by locks, and function more or less the same way. And just like lock modes, latch modes are compatible or incompatible with other latch modes. Table 9-1 shows the latch compatibility matrix and whether the different modes are compatible with each other or not.

Table 9-1. *Latch Compatibility Matrix*

	SH	UP	EX	KP	DT
SH	Yes	Yes	No	Yes	No
UP	Yes	No	No	Yes	No
EX	No	No	No	Yes	No
KP	Yes	Yes	Yes	Yes	No
DT	No	No	No	No	No

Unlike locks, which can partly be controlled by Isolation levels and query hints, latches are completely controlled by the SQL Server engine. This means we cannot modify latch behavior like we can for locks.

Latch Waits

Whenever a latch has to wait because its request couldn't be granted immediately, a latch wait occurs. These waits are tracked and recorded by SQL Server inside the `sys.dm_os_wait_stats` DMV, and also inside a dedicated DMV that records specific latch wait times, `sys.dm_os_latch_waits`, which we will discuss in more detail a bit further down in this chapter.

Figure 9-2 shows a situation in which a latch wait occurs. In this example we are waiting for a data page to be read from the storage subsystem into the buffer cache. In this case latches are used to make sure the same data page on the storage subsystem is not being read into the buffer cache by multiple threads. While the latch is waiting for the page to read into memory, the `PAGEIOLATCH_SH` wait type will be recorded.

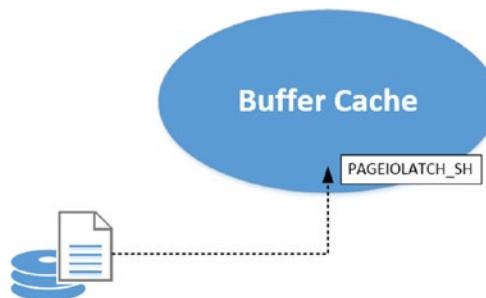


Figure 9-2. *PAGEIOLATCH_SH occurring*

There are three different latch wait types defined in SQL Server that can be accessed by querying the `sys.dm_os_wait_stats` DMV, and they are described in the following list:

- Buffer latches: Buffer latches are used to protect data pages inside the buffer cache. They are not only used for user-related data pages but also for system pages like the Page Free Space (PFS) page that tracks free space inside data pages. Inside the `sys.dm_os_wait_stats` DMV they are indicated by the `PAGELATCH_[xx]` wait type, where the `[xx]` indicates the latch mode used.

- Non-buffer latches: These latches are used to protect data structures outside of the buffer cache. They are indicated by the LATCH_[xx] wait type inside the sys.dm_os_wait_stats DMV.
- IO latches: IO latches are used when data pages are read from the storage subsystem into the buffer cache. This type is indicated by the PAGEIOLATCH_[xx] wait type.

Figure 9-3 shows the number of different latch wait types recorded by the sys.dm_os_wait_stats DMV.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	LATCH_NL	0	0	0	0
2	LATCH_KP	0	0	0	0
3	LATCH_SH	0	0	0	0
4	LATCH_UP	0	0	0	0
5	LATCH_EX	0	0	0	0
6	LATCH_DT	0	0	0	0
7	PAGELATCH_NL	0	0	0	0
8	PAGELATCH_KP	0	0	0	0
9	PAGELATCH_SH	12	0	0	0
10	PAGELATCH_UP	0	0	0	0
11	PAGELATCH_EX	14	0	0	0
12	PAGELATCH_DT	0	0	0	0
13	PAGEIOLATCH_NL	0	0	0	0
14	PAGEIOLATCH_KP	0	0	0	0
15	PAGEIOLATCH_SH	755	797	10	20
16	PAGEIOLATCH_UP	23	7	1	0

Figure 9-3. Latch wait types inside sys.dm_os_wait_stats

Whenever you are looking at the LATCH_[xx] wait type inside the sys.dm_os_wait_stats DMV, you are actually looking at a summary of the wait times for these non-buffer latches. There are various non-buffer latch classes inside SQL Server, and to make it easier to analyze these non-buffer latch classes in more detail, the sys.dm_os_latch_stats DMV was added.

Sys.dm_os_latch_stats

The `sys.dm_os_latch_stats` closely resembles the `sys.dm_os_wait_stats` DMV. The `sys.dm_os_latch_stats` DMV also shows the number of times a wait occurred, the total wait time, and the maximum wait time. The only column missing compared to the `sys.dm_os_wait_stats` DMV is `signal_wait_time_ms`; this is missing because latches do not follow the same execution process (RUNNING, SUSPENDED, RUNNABLE) as requests do.

Figure 9-4 shows a part of the `sys.dm_os_latch_stats` DMV. There are many more non-buffer latch classes, totaling 168, in SQL Server 2017.

	latch_class	waiting_requests_count	wait_time_ms	max_wait_time_ms
1	ACCESS_METHODS_DATASET_PARENT	0	0	0
2	ACCESS_METHODS_HOBT_FACTORY	0	0	0
3	ACCESS_METHODS_HOBT	0	0	0
4	ACCESS_METHODS_HOBT_COUNT	0	0	0
5	ACCESS_METHODS_HOBT_VIRTUAL_ROOT	0	0	0
6	ACCESS_METHODS_CACHE_ONLY_HOBT_ALLOC	0	0	0
7	ACCESS_METHODS_BULK_ALLOC	0	0	0
8	ACCESS_METHODS_SCAN_RANGE_GENERATOR	0	0	0
9	ACCESS_METHODS_KEY_RANGE_GENERATOR	0	0	0
10	ACCESS_METHODS_IOAFF_KEY_RANGE_GENERATOR	0	0	0
11	ACCESS_METHODS_IOAFF_KEY_TARGET_PAGE_CNT	0	0	0
12	ACCESS_METHODS_IOAFF_QUEUE	0	0	0
13	ACCESS_METHODS_IOAFF_READAHEAD_QUEUE	0	0	0
14	ACCESS_METHODS_IOAFF_READAHEAD	0	0	0
15	ACCESS_METHODS_IOAFF_WAITING_WORKER_QUEUE	0	0	0
16	APPEND_ONLY_STORAGE_INSERT_POINT	0	0	0

Figure 9-4. `sys.dm_os_latch_waits`

Just like the `sys.dm_os_wait_stats` DMV, the `sys.dm_os_latch_stats` DMV is cumulative since the start of the SQL Server service. This means it will get reset to 0 value again whenever your SQL Server service is restarted. We can also use the `DBCC SQLPERF` command against the `sys.dm_os_latch_stats` DMV to reset the wait times manually by executing this command:

```
DBCC SQLPERF('sys.dm_os_latch_stats', CLEAR)
```

Page-Latch Contention

One of the most common problems encountered regarding latches is page-latch contention. Page-latch contention occurs when many concurrent latches try to acquire a latch, but there already is a latch in place with an incompatible mode, causing a latch wait. Because this problem can occur on every SQL Server instance that is subjected to concurrent workloads, I want to provide you with the knowledge needed to identify page-latch contention before we discuss the various latch-related wait types.

There are a variety of things that can cause page-latch contention to occur, and even though we have little influence on the latch placement (remember, latches are placed and held by an internal process inside the SQL Server engine), the design of our database can impact latch behavior. One common cause for latch contention is when concurrent queries access so-called hot-spots inside your database. For instance, a small table that holds a few rows that need to be accessed by an application for configuration information can be a potential hot-spot. If many concurrent requests need data from this table, many latches will probably run into other, incompatible latches, causing latch waits to occur and slowing down the application's performance. I have seen this problem occurring various times for different clients, making this a real-world scenario, and I will show you an example of page-latch contention that is based on one of those cases.

In this case the client ran an application that, at specific times, would select large amounts of data and place the results into temporary tables. The application used a large amount of concurrent connections to speed up the creation of these temporary tables. To show the effects of this example, I am going to reproduce the scenario using Ostress to select rows from a table and then insert them into a temporary table.

As input for the Ostress utility, I save a .sql file named `latch_contention.sql` with the query shown in Listing 9-1.

Listing 9-1. Select rows from Sales.SalesOrderDetail into temporary table

```
SELECT TOP (20000) *
INTO #tmptable
FROM Sales.SalesOrderDetail;
```

CHAPTER 9 LATCH-RELATED WAIT TYPES

This query selects the top 20,000 rows from the Sales.SalesOrderDetail table inside the AdventureWorks database and inserts them into a temporary table (#tmpTable).

The next step is to fire up Ostress and execute the latch_contention.sql script with 300 concurrent connections. The Ostress command line I use is shown here:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E  
-dAdventureWorks -i"C:\latch_contention.sql" -n300 -r1 -q
```

While the Ostress utility is running, let's take a look at the sys.dm_os_waiting_tasks DMV to see if anything is running into waits. Figure 9-5 shows a part of the results that are returned when the query that follows is executed:

```
SELECT  
    session_id,  
    wait_duration_ms,  
    wait_type,  
    resource_description  
FROM sys.dm_os_waiting_tasks  
WHERE session_id > 50;
```

	session_id	wait_duration_ms	wait_type	resource_description
1	56	189	PAGELATCH_UP	2:1:1
2	59	171	PAGELATCH_UP	2:1:1
3	57	9	PAGELATCH_UP	2:1:1
4	62	191	PAGELATCH_UP	2:1:1
5	69	183	PAGELATCH_UP	2:1:1
6	108	187	PAGELATCH_UP	2:1:1
7	116	14	PAGELATCH_UP	2:1:1
8	117	198	PAGELATCH_UP	2:1:1
9	118	198	PAGELATCH_UP	2:1:1
10	119	198	PAGELATCH_UP	2:1:1
11	120	198	PAGELATCH_UP	2:1:1
12	122	186	PAGELATCH_UP	2:1:1
13	123	6	PAGELATCH_UP	2:1:1

Figure 9-5. PAGELATCH_UP waits occurring

We are running into a lot of PAGE_{LATCH_UP} waits here that indicate that a latch is waiting to update a page in-memory. The `resource_description` column is very useful here since it indicates the page ID that the latch wants to access. In this case the page ID we are trying to access is 2:3:1. The first number, 2, represents the database ID, which is the TempDB database. The second number, 2, indicates the file ID (the TempDB database on my test system consists of multiple data files). Finally, the last number indicates the page ID, 1. Why are all those sessions waiting on the same wait type against the same data page? This page happens to be a very special page, the Page Free Space (PFS) page. The PFS page tracks how much free space is left inside every page inside the database. It is always the first page of every database (page ID of 1) and has an interval of 8088 pages. So in this example, all the requests are waiting to update the first PFS page inside the TempDB database.

Because we are running inserts into a temporary table using many concurrent connections, we need to find, or allocate, data pages with free space to hold our rows inside the TempDB. All this space usage needs to be updated inside the PFS page, and latches are used to make sure only one thread gets access to the PFS page at a time. Figure 9-6 shows a Perfmon graph of two Perfmon counters, Transaction and Latch Waits/sec. This will show the relationship between to-Ostress workload and the number of latch waits occurring.

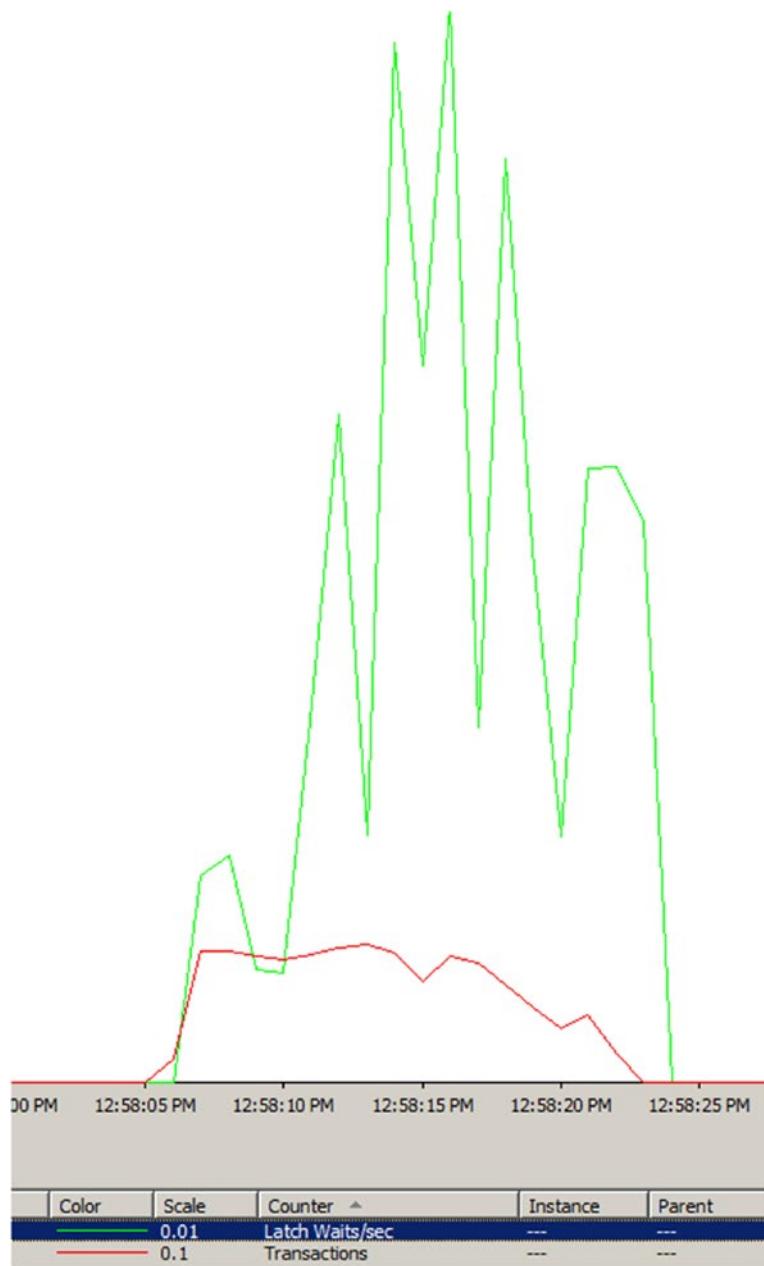


Figure 9-6. Latch Waits/sec and Transactions Perfmon graph

This is actually a classic example of the page-latch contention inside the TempDB database that can occur when many concurrent queries are creating objects inside TempDB. One way to resolve this specific case of latch contention is by adding more (equally sized) TempDB data files. Every new data file will maintain its own PFS pages, and adding more data files helps spread the load of updating the PFS pages. Using the query that follows, I added three more data files to the TempDB database:

```
USE [master]
GO

ALTER DATABASE [tempdb] ADD FILE ( NAME = N'tempdev2', FILENAME =
N'D:\Data\tempdb2.mdf' , SIZE = 204800KB , FILEGROWTH = 10%)
GO

ALTER DATABASE [tempdb] ADD FILE ( NAME = N'tempdev3', FILENAME =
N'D:\Data\tempdb3.mdf' , SIZE = 204800KB , FILEGROWTH = 10%)
GO

ALTER DATABASE [tempdb] ADD FILE ( NAME = N'tempdev4', FILENAME =
N'D:\Data\tempdb4.mdf' , SIZE = 204800KB , FILEGROWTH = 10%)
GO
```

When running the Ostress utility again with the same query to insert data into a temporary table as we did before, I still notice latch waits occurring on PFS pages, but they are spread better across the TempDB data files. When looking at the Transactions and Latch Waits/sec Perfmon counters, I also see fewer latch waits occurring, as shown in Figure 9-7.

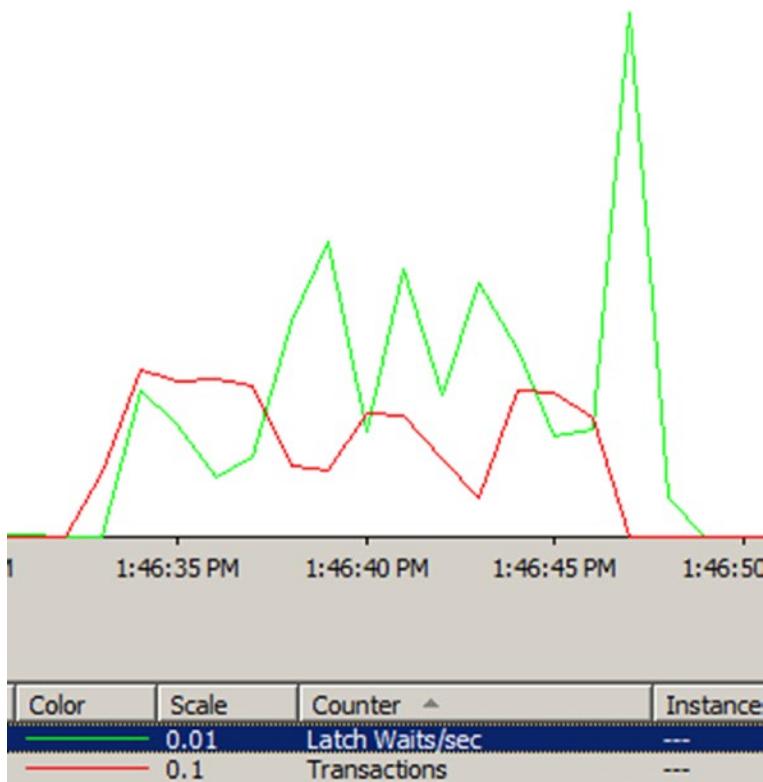


Figure 9-7. Latch Waits/sec and Transaction Perfmon graph after adding more TempDB files

By adding more TempDB data files, I would be able to lower the amount of latch waits even further. Adding too many TempDB data files can be a bad idea though, since the round-robin algorithm that makes sure the data files receive equal allocations can generate noticeable overhead when it needs to manage many TempDB data files. Paul Randal over at his SQLskills blog has a great post discussing TempDB data files and latch contention, which you can find here: www.sqlskills.com/blogs/paul/a-sql-server-dba-myth-a-day-1230-tempdb-should-always-have-one-data-file-per-processor-core/.

Now that we have discussed what latches are and how they work, and looked at an example of latch contention, let's move on and look at latch-related wait types.

PAGELATCH_[xx]

The first latch-related wait type we will discuss in this chapter is the PAGELATCH_[xx] wait type, where the [xx] indicates the latch mode used (e.g., SH for Shared). Since we already discussed the various latch modes in the introduction of this chapter, we won't describe them again in this chapter.

What Is the PAGELATCH_[xx] Wait Type?

PAGELATCH_[xx] waits occur whenever a latch has to wait before it can access a page in-memory. The main cause for these waits is other latches that are already in place on the page and are incompatible with the latch mode our request wants to use. Just like a lock, the latch we want to place on the page has to wait until the incompatible latch is removed from the page. As long as the incompatible latch is in place, our request will record PAGELATCH_[xx] wait time. Figure 9-8 shows a graphical representation of a PAGELATCH_UP wait occurring. I used a cogwheel icon to indicate a latch is already in place on the page to avoid confusion with SQL Server locks.

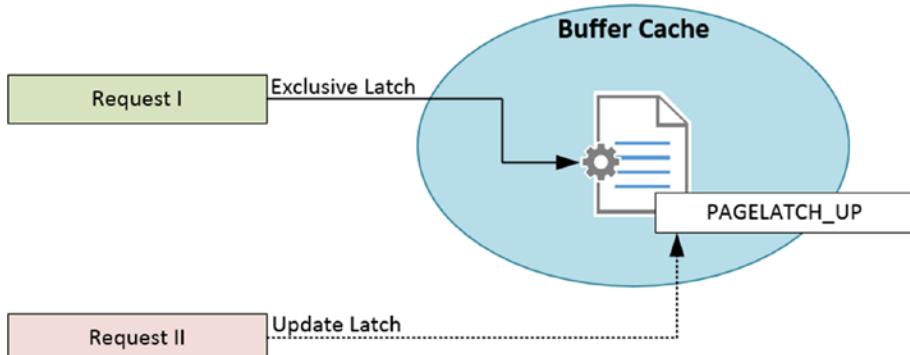


Figure 9-8. PAGELATCH_UP wait occurring

It's easy to confuse PAGELATCH_[xx] waits with PAGEIOLATCH_[xx] waits. Even though they look alike in name, both are completely different latch wait types. The former indicates access to pages already in memory, while the latter indicates pages are being read from disk into memory. We will go into detail regarding the PAGEIOLATCH_[xx] wait type a bit later in this chapter.

PAGELATCH_[xx] Example

In the introduction to this chapter, we took a look at page-latch contention that can occur inside the TempDB database when many concurrent queries are loading data in temporary queries. This isn't the only form of latch contention that can occur inside SQL Server. Another form of latch contention is known as "last-page insert contention." Just like the page-latch contention scenario we discussed earlier, last-page insert contention can also be identified by noticing a high number of PAGELATCH waits. Let's go through an example of last-page insert contention.

Remember that time in database design class when you learned that every table should have a clustered index? And that the best candidate for a clustered index key column is a narrow, unique, ever-increasing value, like an integer? All of that is still true, and it absolutely helps to optimize the performance of queries against those tables. There are, however, very specific cases where using this practice can cause a performance problem known as last-page insert contention.

Last-page insert contention can occur on databases that experience a very heavy insert workload against a table with relatively small rows; for instance, a table with an ID column (Integer data type, auto increasing) and a Name column (Varchar data type). From a best-practice point of view, we would create a clustered index on the ID column since it fits the description of a good index key perfectly. It is narrow, unique for every row, and always increasing. But because of the ever-increasing nature of the auto increment, every newly added row will be added at the end of the clustered index, creating a hot-spot for the last data page of the clustered index. Figure 9-9 shows the insert behavior of rows into data pages inside a clustered index, inside the form of a so-called B-tree structure, which is the data structure SQL Server uses to sort indexes.

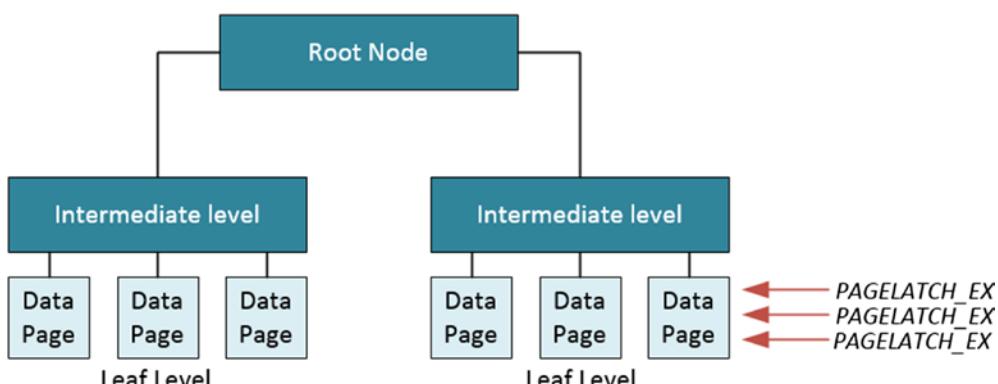


Figure 9-9. Last-page insert contention on last page in a clustered index

Even if the current data page is full, and a new data page is added, the target of the inserts will change to the new page, switching the hot-spot to the new data page.

One question I often hear about this behavior is: “Why aren’t locks stopping this?” The answer is actually pretty simple: because by default we will be using Exclusive row-level locks to insert our new rows instead of locking the page, and you can have multiple concurrent Exclusive row locks on one page. Access to the page that’s in-memory still needs to occur serially though, so latches are used to make sure only one thread has access to the page at any time. Figure 9-10 shows an enlarged view of the data pages at the leaf level of the clustered index with locks in place.

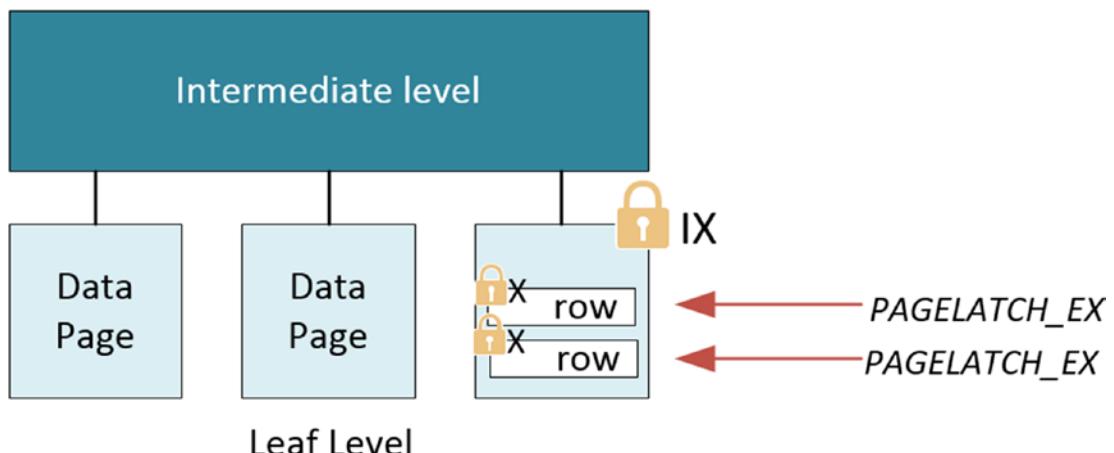


Figure 9-10. Leaf page of clustered index with locks in place

To show you an example of last-page insert contention, I will create a new table inside the AdventureWorks database of my test SQL Server instance using the query that follows:

```
CREATE TABLE Insert_Test
(
    ID INT IDENTITY (1,1) PRIMARY KEY,
    RandomData VARCHAR(50)
);
```

As you can see, this is a pretty small table with an ID column that automatically increases for every new row inserted, and a RandomData column that will hold some data. I indicate that the ID column is the primary key of this table, which will automatically create a clustered index using the ID column as the index key.

The next step is running Ostress with a highly concurrent workload that inserts new rows into the Insert_Test table. This time I don't create a .sql input file for Ostress but rather enter the following query in the Ostress command line:

```
INSERT INTO Insert_Test
  (RandomData)
VALUES
(
  CONVERT(varchar(50), NEWID())
)
```

This will create the following Ostress command line that will connect to the AdventureWorks database and execute the query we supplied using 500 concurrent connections, each connection performing the query 100 times. This should create enough concurrent inserts to demonstrate last-page insert contention:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -Q"INSERT INTO Insert_Test (RandomData) VALUES
(CONVERT(varchar(50), NEWID()))" -n500 -r100 -q
```

While Ostress is running, I query the sys.dm_os_waiting_tasks DMV:

```
SELECT
  session_id,
  wait_duration_ms,
  wait_type,
  resource_description
FROM sys.dm_os_waiting_tasks;
```

This query filters out some columns so that a screenshot of the results will fit on the page. Figure 9-11 shows a portion of the results.

	session_id	wait_duration_ms	wait_type	resource_description
19	338	2	PAGELATCH_EX	5:1:29313
20	492	2	PAGELATCH_EX	5:1:29313
21	433	1	PAGELATCH_EX	5:1:29313
22	468	1	PAGELATCH_EX	5:1:29313
23	552	1	PAGELATCH_EX	5:1:29313
24	422	1	PAGELATCH_EX	5:1:29313
25	502	1	PAGELATCH_EX	5:1:29313
26	471	1	PAGELATCH_EX	5:1:29313
27	470	1	PAGELATCH_EX	5:1:29313
28	389	1	PAGELATCH_EX	5:1:29313
29	414	1	PAGELATCH_EX	5:1:29313
30	381	1	PAGELATCH_EX	5:1:29313
31	334	1	PAGELATCH_EX	5:1:29313
32	387	1	PAGELATCH_EX	5:1:29313
33	462	1	PAGELATCH_EX	5:1:29313
34	501	1	PAGELATCH_EX	5:1:29313
35	402	1	PAGELATCH_EX	5:1:29313

Figure 9-11. PAGELATCH_EX waits on the same page

As expected, the insert workload caused a hot-spot to appear on a page inside the clustered index, in this case the page with a page ID of 29313. All of those tasks shown in Figure 9-11 (and there were around 300 more not shown) are all waiting to place an Exclusive page latch, indicated by the PAGELATCH_EX wait type, on that page so they can perform their insert operation.

To prove that page 29313 is a data page, I am going to use the undocumented DBCC IND command to show us the pages that are associated with the Insert_Test table. DBCC IND will return a row for every page associated with the table we supply as a parameter to DBCC IND, and, among other things, will show us the page type of every page returned.

Running the command that follows will execute the DBCC IND command against the AdventureWorks database's Insert_Test table. Before we run the actual DBCC IND command we have to enable Traceflag 3604 so the results of the DBCC IND command get returned in the SQL Server Management Studio results tab:

```
DBCC TRACEON (3604);
GO
DBCC IND (AdventureWorks, Insert_Test, 1);
GO
```

Figure 9-12 shows the results of the DBCC IND command. Highlighted is page 29313, the page that was the insert hot-spot during the Ostress workload.

	PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
712	1	29302	1	5235	388196433	1	1	72057594062110720	In-row data	1
713	1	29303	1	5235	388196433	1	1	72057594062110720	In-row data	1
714	1	29312	1	5235	388196433	1	1	72057594062110720	In-row data	1
715	1	29313	1	5235	388196433	1	1	72057594062110720	In-row data	1
716	1	29314	1	5235	388196433	1	1	72057594062110720	In-row data	1
717	1	29315	1	5235	388196433	1	1	72057594062110720	In-row data	1
718	1	29316	1	5235	388196433	1	1	72057594062110720	In-row data	1

Figure 9-12. DBCC IND results

The information we are interested in resides in the IndexID and PageType columns. The IndexID column returns the Index ID that this page is associated with. We only have one index on the Insert_Test table, and it has an ID of 1. The PageType column returns the page type of the specific page. In this case the PageType of page 29313 is 1, which indicates that the page is a data page.

Remember, DBCC IND is an undocumented SQL Server command, and I included it here to show you information about the page where the last-page insert contention was occurring. I strongly advise against using it on production servers.

Lowering PAGE[LATCH]_[xx] Waits

So far I have showed you two examples where PAGE[LATCH]_[xx] waits can occur, page-latch contention on the PFS page of the TempDB database and last-page insert contention. There is another latch-contention problem that can occur when inserting rows into a small table with an index. This case of latch contention can also be identified by PAGE[LATCH]_[xx] waits occurring, but it also has a connection with the LATCH_[xx] wait type. For this reason I am saving the explanation and example of this specific case of latch contention for the next section of this chapter where we will discuss the LATCH_[xx] wait type.

Lowering PAGE[LATCH]_[xx] waits can be challenging. Frequently, they are related to the design of your database or your workload, and these can prove difficult to change in production environments. There are, however, a number of factors that can contribute to latch contention that are worth taking the time to check.

It is more common to see latch contention occurring on systems that have a large number of logical processors (16+) and high concurrent OLTP workloads. However, having fewer logical processors does not mean latch contention cannot occur. The examples of latch contention I have shown you so far in this chapter have all been generated on a virtual machine with only two logical processors. I had to create a high enough concurrent workload to reach latch contention. Having more logical processors means there are more threads available to perform work, which also results in more concurrent latches being placed, increasing the chances of latch contention. Adding logical processors when experiencing latch contention can, in this specific case, cause even more latch contention to occur instead of resolving it. Lowering the number of logical processors isn't an option either, because this will slow down all your other workloads.

The best way to resolve latch contention is by identifying where the contention is occurring and what type of contention you are dealing with.

If you are dealing with PFS page contention, a good first step would be to check if you are using one or multiple database data files. If you are using one database data file, the first step would be to add additional, equally sized data files and measure if this lowers the amount of PAGE_{LATCH}_ [xx] waits occurring. If you already have multiple database data files you could try adding more, but be careful not to add too many, because having this can introduce other performance problems. Your goal should be to find a database data file "sweet spot" where you have enough database data files to minimize the impact of latch contention, but not so many as to cause the overhead to become too high. This depends entirely on your workload, so it is impossible for me to give you a generalized recommendation.

When dealing with last-page insert contention you could consider changing the index key to something else instead of a sequentially increasing value, like a GUID. Using a GUID as an index key will result in a larger index because of the byte requirements of a GUID. Also, because GUIDs are entirely random, keeping the index in order requires more work than when dealing with an ever-increasing, sequential value. It can also have consequences for your applications or queries that possibly would need to be rewritten to accommodate the change in data type.

Other factors to consider that can impact latch contention are indexing strategies, page fullness, and the number of concurrent connections to the database. Also, identifying and optimizing the access patterns to the data inside the database can help immensely. For instance, if you know your workload consists of many very small inserts

against a single table, it might be worth taking the time to see if you can combine some of the small inserts into a larger batch, effectively lowering the number of latches needed.

One final option for resolving latch contention is using a method called hash partitioning. Hash partitioning splits up your table or index into various partitions based on a value that is generated by using a computed column. Partitioning is only available in Enterprise Edition (unless you are running SQL Server 2016 SP1 or higher, in that case table and index partitioning is also available in Standard Edition), but it is a method that can minimize, or completely prevent, latch contention.

Hash partitioning works by cutting up tables or indexes into partitions, with each partition holding a set of the data. Partitioning is frequently used for archiving data from inside a table to another filegroup that resides on other (cheaper) storage, while the current data resides on fast storage. In the case of hash partitioning, we are going to calculate a value for every row inside the table using a computed column. Based on that value, we will move the row to a partition.

The great advantage of using this method of partitioning indexes is that every partition has its own index tree. So even though the insert statements will still occur on the last (right-most) page of the index, it will be spread across the partitions. If we look at Figure 9-13, we see a case of last-page insert contention, where concurrent queries are trying to insert rows into an index as we discussed in the preceding example.

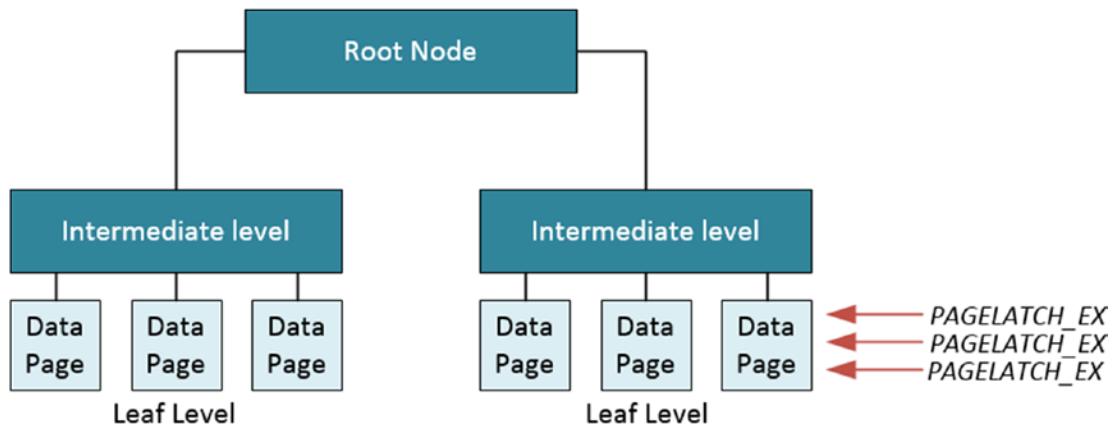


Figure 9-13. Last-page insert contention on the right-most data page of an index

If we were to use partitioning to cut the index into multiple parts (three in this case), we would get the situation shown in Figure 9-14, multiple B-trees, each spanning a part of the data.

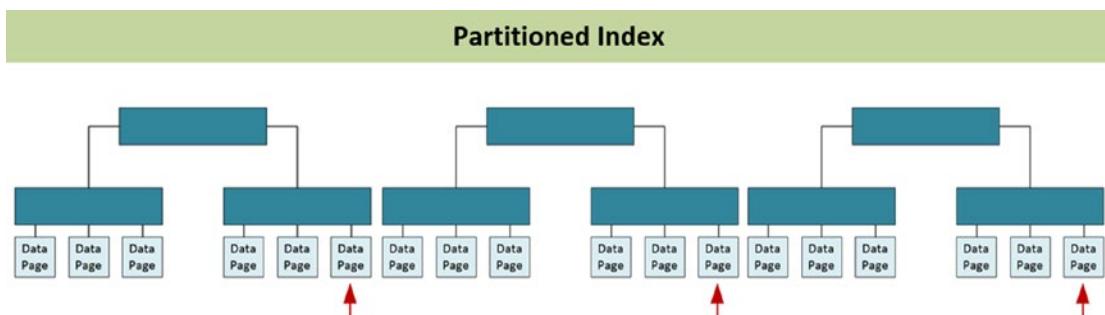


Figure 9-14. Last-page inserts spread across partitions

Let's take a look at the effects of hash partitioning when we run the workload to generate last-page insert contention, like we did in the example earlier in this chapter.

The first thing we need to do is create a Partition function. This will map rows inside the table or index to partitions based on the value of a column. The following script will create a Partition function named LatchPartFunc that will divide rows into nine partitions based on the value of a column (which we will create a bit later). See the following:

```
CREATE PARTITION FUNCTION [LatchPartFunc] (INT)
AS RANGE LEFT FOR VALUES
(0,1,2,3,4,5,6,7,8);
```

The next step is to create a Partition scheme that will map the partitions to a filegroup:

```
CREATE PARTITION SCHEME [LatchPartSchema]
AS PARTITION [LatchPartFunc] ALL TO ([PRIMARY]);
```

In this case I used the PRIMARY filegroup, but you are free to create an additional filegroup to hold the partitions.

Next up is creating a new table called Insert_Test3 using the query that follows. Notice the ID_Hash column. This is a computed column that will calculate a value between 0 and 8 based on the value of the ID column:

```
CREATE TABLE Insert_Test3
(
    ID INT IDENTITY(1,1),
    RandomData VARCHAR(50),
```

```
ID_Hash AS (CONVERT(INT, abs(binary_checksum(ID) % (9)), (0))) PERSISTED
);
```

The last step is to create a clustered index and map it to the Partition scheme:

```
CREATE UNIQUE CLUSTERED INDEX idx_ID
ON Insert_Test3
(
ID ASC, ID_Hash
)
ON LatchPartSchema(ID_Hash);
```

Now that we have our partitioned table in place, let's repeat our Ostress workload that caused last-page insert contention in our previous example. I changed the target table for the inserts to our new, partitioned `Insert_Test3` table.

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -Q"INSERT INTO Insert_Test3 (RandomData) VALUES
(CONVERT(varchar(50), NEWID()))" -n500 -r100 -q
```

During both Ostress workloads I used Perfmon to monitor the number of latch waits occurring every second. Figure 9-15 shows the Perfmon graph for the first Ostress workload against a non-partitioned index and the second against the partitioned index we just created.

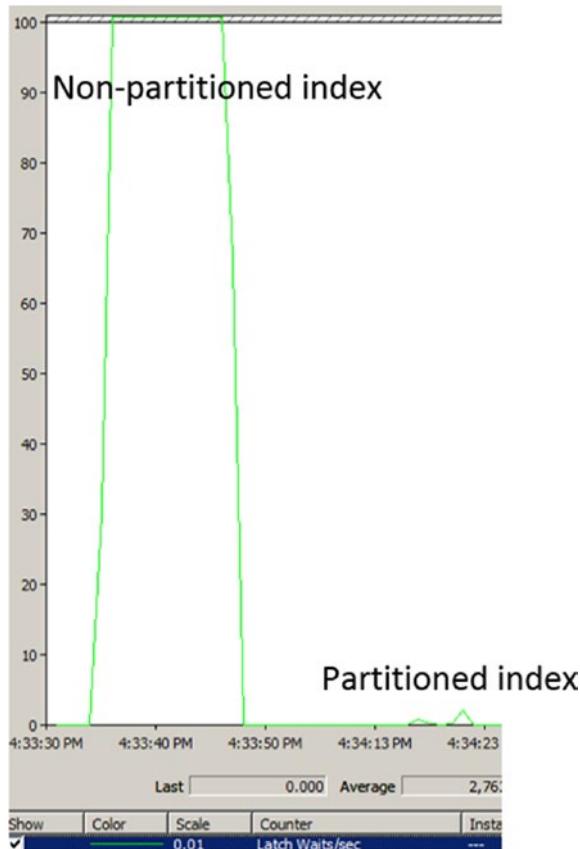


Figure 9-15. Latch Waits/sec against both a non-partitioned and a partitioned index

As you can see, the number of latch waits occurring dropped drastically after configuring hash partitioning! We can view the distribution of rows across the different partitions we created by running this query:

```
SELECT *
FROM sys.partitions
WHERE object_id = OBJECT_ID('Insert_Test3');
```

Figure 9-16 shows the results of this query on my test SQL Server instance.

	partition_id	object_id	index_id	partition_number	hobt_id	rows	filestream_filegroup_id	data_compression	data_compression_desc
1	72057594062241792	420196547	1	1	72057594062241792	5555	0	0	NONE
2	72057594062307328	420196547	1	2	72057594062307328	5556	0	0	NONE
3	72057594062372864	420196547	1	3	72057594062372864	5556	0	0	NONE
4	72057594062438400	420196547	1	4	72057594062438400	5556	0	0	NONE
5	72057594062503936	420196547	1	5	72057594062503936	5556	0	0	NONE
6	72057594062569472	420196547	1	6	72057594062569472	5556	0	0	NONE
7	72057594062635008	420196547	1	7	72057594062635008	5555	0	0	NONE
8	72057594062700544	420196547	1	8	72057594062700544	5555	0	0	NONE
9	72057594062766080	420196547	1	9	72057594062766080	5555	0	0	NONE
10	72057594062831616	420196547	1	10	72057594062831616	0	0	0	NONE

Figure 9-16. Rows distribution across partitions

In Figure 9-16 we see the nine partitions we created on the Insert_Test3 table, numbered 1 to 9 by the partition_number column. The rows column shows the number of rows inside each partition, and as you can see, they are distributed very evenly across the nine partitions! The hobt_id returns the ID of the B-tree where two rows of this partition are stored; all the partitions have different IDs, meaning they each have their own B-tree structure.

Even though partitioning is a great way to resolve latch contention issues, it does come with its own unique challenges and drawbacks. Two of those are that it is an Enterprise-only feature (unless you are on SQL Server 2016 SP1 or higher), thus costly, and it can impact the generation of query execution plans, resulting in a suboptimal plan.

PAGELATCH_[xx] Summary

The PAGELATCH_[xx] wait type indicates that buffer latches, which are used to protect in-memory pages, are running into other, non-compatible, buffer latches. Just like locks, latches have different modes they use when protecting pages, and not all of these are compatible with each other. Seeing a large amount of PAGELATCH_[xx] waits occurring can indicate a case of latch contention. Resolving latch contention can be challenging and frequently requires making changes to the database design or queries.

LATCH_[xx]

Another latch-related wait type is the LATCH_[xx] wait type. Just like the PAGELATCH_[xx] wait type we discussed in the previous section, LATCH_[xx] waits are related to a specific latch class. While the PAGELATCH_[xx] wait type is related to latches that protect data structures inside the buffer cache, the LATCH_[xx] wait type is related to latches that are used to protect data structures outside of the buffer cache (but still inside the SQL Server memory).

What Is the LATCH_[xx] Wait Type?

When you see the LATCH_[xx] wait type occurring a specific class of non-buffer latches is running into a wait. The LATCH_[xx] is actually a summary of the wait time of those different non-buffer latch classes and not a latch type of its own. All of the different non-buffer latch classes that add to the wait time shown by the LATCH_[xx] wait type are recorded inside their own DMV, sys.dm_os_latch_stats. There are many different latch classes that the LATCH_[xx] wait type represents, totaling 168 in SQL Server 2017. Figure 9-17 shows the memory area where LATCH_[xx] waits can occur.

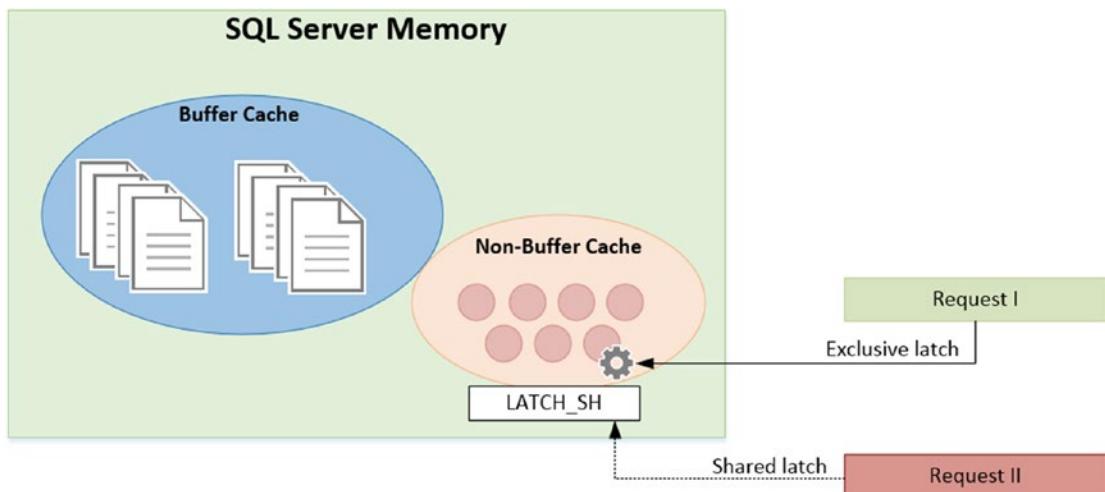


Figure 9-17. LATCH_SH wait occurring

Because the LATCH_[xx] wait type is a cumulative view of waits occurring on a specific latch class, you will need to look inside the sys.dm_os_latch_stats DMV to find the exact cause of the LATCH_[xx] wait. We described the inner workings and columns of the sys.dm_os_latch_stats DMV in the “Introduction to Latches” section at the start of this chapter, so I won’t go into more detail about the DMV here.

LATCH_[xx] Example

There is one case of latch contention that can occur that will result in LATCH_[xx] waits. This problem can occur on small tables that have a shallow B-tree structure (we will explain more about the B-tree structure a bit further down in this section) during a large

volume of concurrent insert operations. A typical use case of such a table could be a messaging table that acts as a queue and gets truncated when the messages are sent. The script in Listing 9-2 will create a test table, named Insert_Test2, together with a non-clustered index on the table.

Listing 9-2. Test contention table with non-clustered index

```
-- Create the table
CREATE TABLE Insert_Test2
(
    ID UNIQUEIDENTIFIER,
    RandomData VARCHAR(50)
);

-- Create a non-clustered index on the ID column
CREATE NONCLUSTERED INDEX idx_ID
ON Insert_Test2 (ID);
GO
```

The ID column has a data type of UNIQUEIDENTIFIER to make sure random, non-sequential values are generated. By creating a non-clustered index on this column, we are sure inserts will happen randomly across the B-tree associated with the non-clustered index.

Once the table is created we can start Ostress with a workload consisting of an insert query that will insert a single row inside the table. We will run the workload with 500 concurrent connections, with each of the connections executing the query 100 times. The command that follows shows the Ostress command:

```
"C:\Program Files\Microsoft Corporation\RMLUtils\ostress.exe" -E
-dAdventureWorks -Q"INSERT INTO Insert_Test2 (ID, RandomData) VALUES
(NEWID(), CONVERT(varchar(50), NEWID()))" -n500 -r100 -q
```

While the workload is running, I can take a look at the sys.dm_os_waiting_tasks DMV using the following query so the resource_description column could fit on the screenshot:

```
SELECT
    session_id,
    wait_duration_ms,
```

```

wait_type,
resource_description
FROM sys.dm_os_waiting_tasks;

```

Figure 9-18 shows a part of the results of this query on my test SQL Server instance.

	session_id	wait_duration_ms	wait_type	resource_description
13	224	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
14	214	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
15	343	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
16	427	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
17	430	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
18	325	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
19	292	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
20	188	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
21	391	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
22	341	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
23	393	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
24	299	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
25	394	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
26	174	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
27	395	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
28	389	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)
29	385	10	LATCH_SH	ACCESS_METHODS_HOBT_VIRTUAL_ROOT (00000020C0790AF8)

Figure 9-18. LATCH_SH waits occurring

The resource_description column of the sys.dm_os_waiting_tasks DMV will help us identify what latch class is associated with the LATCH_[xx] wait. In this case we are running into the ACCESS_METHODS_HOBT_VIRTUAL_ROOT latch class.

Now that we know what latch class is running into waits, we can query the sys.dm_os_latch_waits DMV to find out the number of waits and the total wait time for this specific latch class using the following query:

```

SELECT *
FROM sys.dm_os_latch_stats
WHERE latch_class = 'ACCESS_METHODS_HOBT_VIRTUAL_ROOT';

```

Figure 9-19 shows the results of this query on my test SQL Server instance.

latch_class	waiting_requests_count	wait_time_ms	max_wait_time_ms
1 ACCESS_METHODS_HOBT_VIRTUAL_ROOT	44407	636426	297

Figure 9-19. ACCESS_METHOD_HOBT_VIRTUAL_ROOT latch wait

Now that we have identified the latch class that is causing the LATCH_SH wait to occur we can start troubleshooting it. According to Books Online, the ACCESS_METHODS_HOBT_VIRTUAL_ROOT latch class is used to “synchronize access to the root page abstraction of an internal B-tree.” Even though the description is pretty limited, it should give us an idea of where to start looking when troubleshooting this specific problem. I included a list of the different latch classes that are described on Books Online, including some extra information whenever possible, in Appendix III of this book.

Apparently, something happened to the B-tree structures associated with indexes. Since we only have one non-clustered index on the table we created (idx_ID), we can assume something happened to the B-tree of that index. Let’s refresh our memory a little bit about what a B-tree index structure looks like by looking at Figure 9-20.

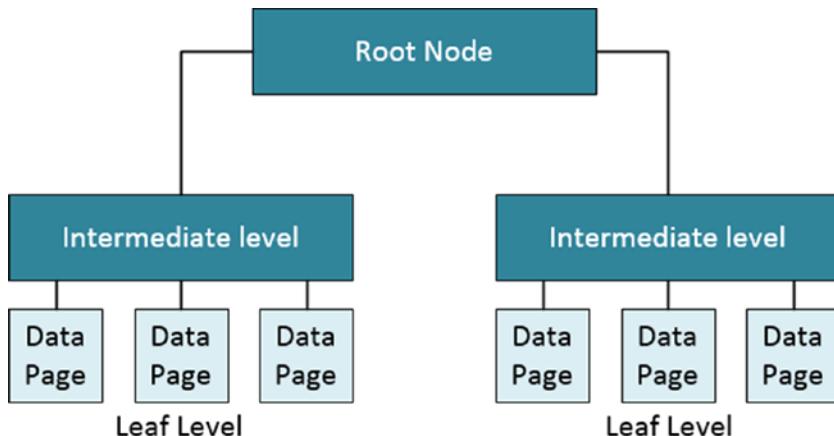


Figure 9-20. B-tree index structure

The B-tree structure we see in Figure 9-20 is pretty shallow, as it only has three levels. The first one is the Root Node (level 0), the second is the Intermediate level (level 1), and finally at the bottom of the B-tree are the data pages that hold the actual index keys (or in case of a clustered index, the entire row). In Figure 9-20 we only have one level of intermediate nodes, but depending on the number of data pages inside the index it is possible to have more Intermediate levels. When the table is very small it is possible to have the data pages inside the Intermediate level instead of a level further down the B-tree.

Whenever SQL Server needs to navigate through the B-tree it will start at the Root page inside the Root Node. The Root page will help it navigate down to the index page that holds the information it needs inside the Intermediate level. In turn, the

Intermediate page can send the request further down the B-tree if the page is not inside the Intermediate level, but is rather at the Leaf level. The Leaf level is the last level in an index; it cannot navigate further down than that. Figure 9-21 shows how SQL Server navigates through the B-tree. In this case I used numbers as the index key to make it a bit easier.

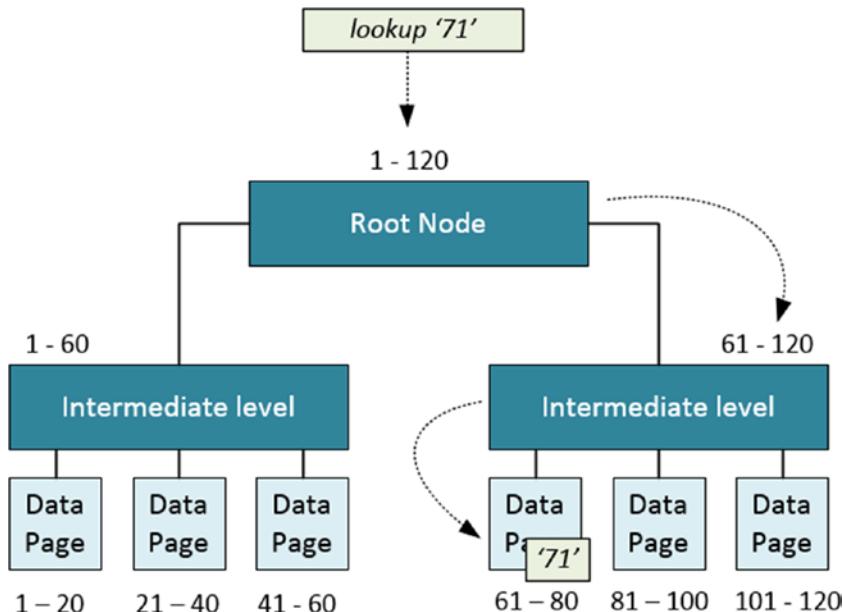


Figure 9-21. B-tree navigation

When data is added to the index, the index will allocate new data pages at the Leaf level to hold the index keys (remember, clustered indexes hold rows at the Leaf level). Whenever enough data is inserted that a new index level needs to be created, a Root page split occurs so the new level inside the index can be accessed. This Root page split doesn't cut your Root page into two new ones, there is always one Root page, but it needs to be updated so we can use it to navigate through the B-tree to the new data.

The ACCESS_METHODS_HOBT_VIRTUAL_ROOT latch class is the latch wait class that is associated whenever a Root page split occurs in order to create another level inside the B-tree. Whenever a Root page split occurs, the B-tree will acquire an Exclusive latch. All threads that want to navigate down the B-tree will have to wait for the Root page split to finish since they use Shared latches that are incompatible with the Exclusive latch. But why are we seeing LATCH_SH waits occurring when running our Ostress workload,

instead of seeing Exclusive latches, since we are performing inserts? The reason for that is pretty simple: before SQL Server knows where to place the new index key inside the index, it first has to navigate through the B-tree to locate where the new index key needs to be placed, and it uses Shared latches during its navigation.

To show you that another level was added to the non-clustered index during our Ostress workload, I am going to use the INDEXPROPERTY function to retrieve the depth of the non-clustered index we created.

The first thing I am going to do is empty our Insert_Test2 table using the TRUNCATE command:

```
TRUNCATE TABLE Insert_Test2;
```

If we use the INDEXPROPERTY function against the non-clustered index on this table, we can view the current depth of the B-tree. The query that follows shows how to use the INDEXPROPERTY function to retrieve this information:

```
SELECT INDEXPROPERTY(OBJECT_ID('Insert_Test2'), 'idx_ID', 'indexDepth')
```

Since we just truncated the table, the index depth should be 0 as there are no rows inside the table yet.

I then run the Ostress workload again, and after it has finished I look at the index information again. Instead of using the INDEXPROPERTY function, I use the sys.dm_db_index_physical_stats DMF to return some additional information about the number of index and data pages inside the index. This query returns such information:

```
SELECT
    index_id,
    index_type_desc,
    index_depth,
    index_level,
    page_count,
    record_count
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'AdventureWorks'), OBJECT_ID(N'Insert_Test2'), NULL, NULL ,
    'DETAILED');
```

Figure 9-22 shows the results of this query on my test SQL Server instance.

	index_id	index_type_desc	index_depth	index_level	page_count	record_count
1	0	HEAP	1	0	425	50000
2	2	NONCLUSTERED INDEX	3	0	261	50000
3	2	NONCLUSTERED INDEX	3	1	2	261
4	2	NONCLUSTERED INDEX	3	2	1	2

Figure 9-22. *sys.dm_db_index_physical_stats results*

As you can see in this image, the non-clustered index now has three levels as indicated by the `index_depth` column. The `index_level` and `page_count` columns show how many pages exist on each level of the B-tree. The highest `index_level` number is the Root level, the lowest the Leaf level.

While the new levels were created inside the B-tree, the concurrent insert queries had to wait before they could navigate the B-tree, resulting in the `LATCH_SH` waits.

Lowering LATCH_[xx] Waits

In the previous example I presented a specific case of latch contention that occurs when index Root page splits occur so as to extend the B-tree structure. As I mentioned before, there are many, many more latch classes that are reported by the `LATCH_[xx]` wait type. This makes describing “one-size-fits-all” suggestions impossible. I can describe a general approach though, using the list here:

- Query `sys.dm_os_waiting_tasks` if `LATCH_[xx]` waits are occurring. The `resource_description` column can show you additional information about the specific latch class. If you are in a situation where the `LATCH_[xx]` waits do not show in `sys.dm_os_waiting_tasks` but high wait times are visible in `sys.dm_os_wait_stats` DMV, the `sys.dm_os_latch_waits` DMV should be your starting point.
- Another helpful DMV can be the `sys.dm_exec_requests` DMV. Joined together with the `sys.dm_exec_sql_text` DMF, it may help you to find the query that is causing the `LATCH_[xx]` wait.
- Query `sys.dm_os_latch_waits` to see if this correlates with the latch class shown in the `resource_description` column of the `sys.dm_os_waiting_tasks` DMV.

- Check Books Online or Appendix III in this book for more information about the specific latch class.

Another good resource for more information about common latch classes is Paul Randal's blog post "Most common latch classes and what they mean" at www.sqlskills.com/blogs/paul/most-common-latch-classes-and-what-they-mean/. Though Paul only describes the ten most common latch classes, it can be a good starting point for your investigation.

Thankfully, it is not very common to see consistent high wait times for the LATCH_[xx] wait type since the cases that can cause the LATCH_[xx] waits to occur are frequently related to very specific workloads and database design.

LATCH_[xx] Summary

The LATCH_[xx] wait type represents waits encountered by a large selection of different, non-buffer-related latch classes inside SQL Server. These non-buffer-related latch classes have their own latch wait DMV, `sys.dm_os_latch_waits`, that returns the wait times of those latch classes. Troubleshooting LATCH_[xx] waits can be difficult since the latch classes that are associated with the wait type are minimally documented. Thankfully, it is not very common to see high wait times on the LATCH_[xx] wait type since they only occur for very specific situations and workloads.

PAGEIOLATCH_[xx]

The final latch-related wait type we will discuss in this chapter is the PAGEIOLATCH_[xx] wait type. The PAGEIOLATCH_[xx] wait type is by far the most common latch-related wait type and together with the CXPACKET wait type is the most common wait type to see on any SQL Server instance.

Just like the two previous latch wait types we discussed, the PAGEIOLATCH_[xx] has different access modes that I replaced with [xx] in this chapter. Since we already described the different latch modes in the introduction, we won't discuss them further in this chapter.

So far we have discussed two of the three latch-related wait types and the areas they are related to. The PAGELATCH_[xx] wait type was related to latches being placed on memory pages inside the buffer cache, and the LATCH_[xx] wait type is related to latches on non-buffer objects. The PAGEIOLATCH_[xx] wait type also indicates the use of latches on a specific area in SQL Server, in this case the IO latches.

What Is the PAGEIOLATCH_[xx] Wait Type?

Disk operations inside SQL Server are very expensive. Accessing the disk subsystem of your system requires extra resources and is always slower than accessing information that is inside the memory of your system. Because SQL Server is a database, and accessing and storing data inside the database is its primary function, the way SQL Server accesses data is extremely important. If data access is slow, SQL Server will perform slower as well, and this can result in noticeable performance degradation inside your queries or applications. To make IO interactions as efficient as possible, SQL Server uses a buffer cache to cache data pages that were previously accessed into the memory of your system. By caching data pages SQL Server only has to access the disk subsystem once, when the first query requests those specific data pages. When later queries require the same data pages as the first query, SQL Server will detect that those pages are already inside the buffer cache, through the Buffer Manager, and will access the data pages from inside the buffer cache instead of performing extra interactions with the disk subsystem. Figure 9-23 shows the buffer cache behavior when a query requires data pages from the storage subsystem.

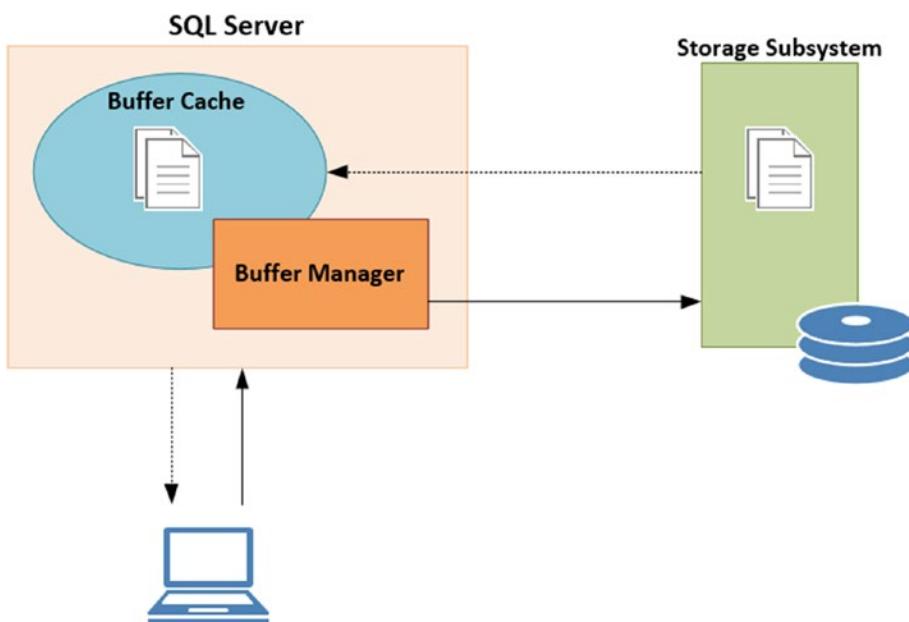


Figure 9-23. Moving a page from the storage subsystem to the buffer cache

During the movement of data pages from the storage subsystem to the buffer cache, latches are used to “reserve” a buffer page for the data page on the storage subsystem. This makes sure no other concurrent transactions allocate the same buffer page, or simultaneously attempt to transfer the same data page from the storage subsystem to the buffer cache.

While SQL Server is transferring the data page from the storage subsystem into the buffer cache, an Exclusive latch will be placed on the buffer page. Because Exclusive latches are incompatible with almost every other latch mode (save for the Keep mode), it is guaranteed that no other latch can access the buffer page while it is being transferred. From the user perspective, a PAGEIOLATCH_[xx] wait will be recorded for the duration of the transfer of the data page. The mode of the latch depends on the action that initiated the movement of the data page from the storage subsystem to the buffer cache. A PAGEIOLATCH_SH will be recorded if the data is being moved for read access, and a PAGEIOLATCH_UP or PAGEIOLATCH_EX will be used if the data page is being moved for a modification. Figure 9-24 shows the data page movement including latches and latch waits.

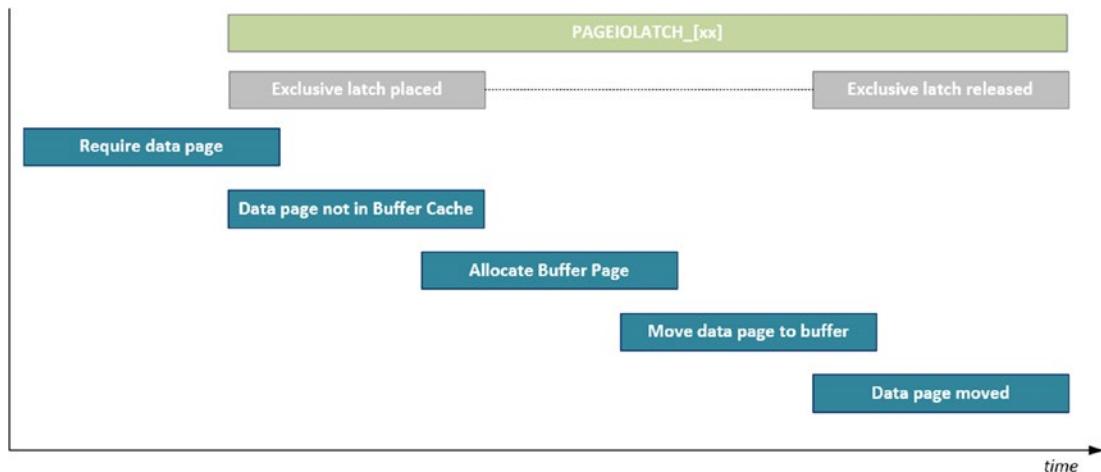


Figure 9-24. Movement of data page

To summarize the preceding section, if you see PAGEIOLATCH_[xx] waits occurring, it means your SQL Server instance is reading data from your storage subsystem into your buffer cache. Because this is a very common operation to perform, it is easy to see why the PAGEIOLATCH_[xx] wait type is one of the most common wait types on any SQL Server instance.

PAGEIOLATCH_[xx] Example

Creating an example for PAGEIOLATCH_[xx] waits is extremely easy—just run a SELECT query against a freshly restarted SQL Server instance. A restart of the SQL Server service will empty the buffer cache of all data pages. This will leave you with a buffer cache without any user data inside it. There is, however, another way to clear the buffer cache without needing to restart the SQL Server service. Running the DBCC DROPCLEANBUFFERS command will remove all the unmodified data pages from the buffer cache. Combining it with the CHECKPOINT command will ensure the modified pages are also written to disk, leaving you with an empty, or “cold,” buffer cache.

The query in Listing 9-3 will perform a CHECKPOINT, followed by a DBCC DROPCLEANBUFFERS. It will then reset the sys.dm_os_wait_stats DMV and run a query against the AdventureWorks database. After the query against some of the tables inside the AdventureWorks database, we will query the sys.dm_os_wait_stats DMV for PAGEIOLATCH_[xx] waits.

Listing 9-3. Generate PAGEIOLATCH_SH waits

```
CHECKPOINT 1;
GO

DBCC DROPCLEANBUFFERS;
GO

DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
GO

SELECT
    SOD.SalesOrderID,
    SOD.CarrierTrackingNumber,
    SOH.CustomerID,
    C.AccountNumber,
    SOH.OrderDate,
    SOH.DueDate
FROM Sales.SalesOrderDetail SOD
INNER JOIN Sales.SalesOrderHeader SOH
ON SOD.SalesOrderID = SOH.SalesOrderID
```

```

INNER JOIN Sales.Customer C
ON SOH.CustomerID = C.CustomerID;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'PAGEIOLATCH_%';

```

Figure 9-25 shows the results of the last query against the `sys.dm_os_wait_stats` DMV on my test SQL Server instance.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PAGEIOLATCH_NL	0	0	0	0
2	PAGEIOLATCH_KP	0	0	0	0
3	PAGEIOLATCH_SH	33	19	1	0
4	PAGEIOLATCH_UP	0	0	0	0
5	PAGEIOLATCH_EX	0	0	0	0
6	PAGEIOLATCH_DT	0	0	0	0

Figure 9-25. PAGEIOLATCH_SH wait time information

It makes sense to see only PAGEIOLATCH_SH waits recorded, since we are executing a SELECT query instead of performing data modification. If we were to perform some form of data modification against the rows, we would see PAGEIOLATCH_EX waits in the results.

Lowering PAGEIOLATCH_[xx] Waits

If you read the previous section, you would probably understand that seeing PAGEIOLATCH_[xx] waits occur is completely normal behavior inside SQL Server. In many cases your databases are larger in size than the available amount of RAM inside your system, and some interaction with the storage subsystem is to be expected. Even if your databases are smaller than the amount of RAM in your system, and they can fit entirely inside the buffer cache, you will still notice PAGEIOLATCH_[xx] waits occurring during the startup of SQL Server, since this is the time SQL Server will start moving data pages from the storage subsystem into the buffer cache (if there is any query activity, SQL Server won't move data from the storage subsystem into the buffer cache by itself).

Since seeing PAGEIOLATCH_[xx] waits occur is completely normal for every SQL Server instance, it is very important to maintain a baseline of the wait times (Chapter 4, “Building a Solid Baseline,” can help you with that). When the wait times stay within the range of the baseline values for this wait type, there shouldn’t be any cause for concern. If wait times are much higher than you expect them to be, investigation into the source of the higher-than-normal wait times might be necessary. There are quite a few possible causes for seeing higher-than-normal PAGEIOLATCH_[xx] wait times, and I will describe some of the more common ones.

The first place I look when noticing higher-than-normal PAGEIOLATCH_[xx] wait times is the SQL Server log to find out if SQL Server was restarted. SQL Server can restart due to a crash, but also when a failover occurs. These events will cause high PAGEIOLATCH_[xx] wait times that might not be reflected in your baseline, especially when SQL Server restarts do not frequently occur. Since our baseline’s measurements are frequently calculated using average values, the PAGEIOLATCH_[xx] wait times during SQL Server startup slowly lower when more measurements are taken inside the average baseline. If you create your baseline on measurements taken between a specific time range, and SQL Server hasn’t had a restart during the time range, your baseline measurements will also be considerably lower. As we read in the example section, a DBCC DROPCLEANBUFFERS will also remove data pages from the buffer cache, resulting in higher PAGEIOLATCH_[xx] wait times after the command completes. Sadly, unlike the DBCC FREEPROCCACHE command, the execution of the DBCC DROPCLEANBUFFERS command is not recorded in the SQL Server log.

One of the more common pieces of advice I see about lowering PAGEIOLATCH_[xx] wait times is to focus your attention on the storage subsystem. Since the PAGEIOLATCH_[xx] wait type indicates data movement from your storage subsystem to your buffer cache, it is logical that the storage subsystem plays a vital role in the wait times, but do not automatically assume this is the root cause! If you do have storage-related problems, this can show in the PAGEIOLATCH_[xx] wait time, so checking the performance of your storage subsystem is worth the effort.

A good place to start for monitoring storage performance is Perfmon. Perfmon has a variety of counters that will show you the current performance of your storage subsystem. Those in the following list are the ones I use the most when monitoring storage performance:

- **PhysicalDisk\Avg. Disk sec/Read:** This will show you the average read latency on the disk you are monitoring. Less latency is better, and as a general guideline latency values should be below 20 milliseconds (0.020 within Perfmon, as it reports the latency in seconds).
- **PhysicalDisk\Avg. Disk sec/Write:** This will return the average write latency on the disk you are monitoring. Just like the read latency, write latency should, as a general guideline, be below 20 milliseconds.
- **PhysicalDisk\Disk Reads/sec:** This shows the amount of read IOPS (Input Output Operations) per second. This information can be helpful if you are running into capacity issues on the disk.
- **PhysicalDisk\Disk Writes/sec:** The same as the **PhysicalDisk\Disk Reads/sec**, but this one shows the amount of write IOPS.
- **PhysicalDisk\Disk Read Bytes/sec:** This counter shows the amount of bytes read from the disk per second. Again, this information can be useful for detecting possible capacity problems.
- **PhysicalDisk\Disk Write Bytes/sec:** This is identical to the **PhysicalDisk\Disk Read Bytes/sec**, but this counter shows the amount of bytes written to disk per second.

Using the information these Perfmon measurements provide, you should be able to identify possible storage-related bottlenecks. This information can also be helpful to the storage administrator (if there is one) who can compare these measurements to the measurements of the storage he/she manages.

As an extra diagnostic tool, or if you cannot use Perfmon, you can also run the IO performance script Paul Randal created based on the `sys.dm_io_virtual_file_stats` DMF, shown in Listing 9-4. The script, and the blog post describing the script, can be found on Paul's blog at www.sqlskills.com/blogs/paul/how-to-examine-io-subsystem-latencies-from-within-sql-server/.

Listing 9-4. IO performance script

```

SELECT
    [ReadLatency] =
        CASE WHEN [num_of_reads] = 0
            THEN 0 ELSE ([io_stall_read_ms] / [num_of_reads]) END,
    [WriteLatency] =
        CASE WHEN [num_of_writes] = 0
            THEN 0 ELSE ([io_stall_write_ms] / [num_of_writes]) END,
    [Latency] =
        CASE WHEN ([num_of_reads] = 0 AND [num_of_writes] = 0)
            THEN 0 ELSE ([io_stall] / ([num_of_reads] + [num_of_writes])) END,
    [AvgBPerRead] =
        CASE WHEN [num_of_reads] = 0
            THEN 0 ELSE ([num_of_bytes_read] / [num_of_reads]) END,
    [AvgBPerWrite] =
        CASE WHEN [num_of_writes] = 0
            THEN 0 ELSE ([num_of_bytes_written] / [num_of_writes]) END,
    [AvgBPerTransfer] =
        CASE WHEN ([num_of_reads] = 0 AND [num_of_writes] = 0)
            THEN 0 ELSE
                ((([num_of_bytes_read] + [num_of_bytes_written]) /
                ([num_of_reads] + [num_of_writes]))) END,
LEFT ([mf].[physical_name], 2) AS [Drive],
DB_NAME ([vfs].[database_id]) AS [DB],
[mf].[physical_name]
FROM
    sys.dm_io_virtual_file_stats (NULL,NULL) AS [vfs]
JOIN sys.master_files AS [mf]
    ON [vfs].[database_id] = [mf].[database_id]
    AND [vfs].[file_id] = [mf].[file_id]
-- WHERE [vfs].[file_id] = 2 -- log files
ORDER BY [Latency] DESC
-- ORDER BY [ReadLatency] DESC
-- ORDER BY [WriteLatency] DESC;
GO

```

Figure 9-26 shows a part of the results of the IO performance script from Listing 9-4 on my test SQL Server instance, ordered by latency.

	ReadLatency	WriteLatency	Latency	AvgBPerRead	AvgBPerWrt	AvgBPerTransfer	Drv	DB	physical_name
1	6	0	4	93090	2048	64640	C:	IO_test	C:\SQL\Log\IO_test_log.ldf
2	4	0	3	48593	2662	40087	C:	trans_demo	C:\SQL\Log\trans_demo.ldf
3	3	1	2	93090	2048	64640	C:	Baseline	C:\SQL\Log\baseline_log.ldf
4	1	0	1	59517	8192	58490	C:	IO_test	C:\SQL\Data\IO_test.mdf
5	1	1	1	122138	154808	144570	C:	AdventureWorks	C:\SQL\Data\AdventureWorks2014_Data.mdf
6	1	1	1	58982	8192	57878	C:	trans_demo	C:\SQL\Data\trans_demo.mdf
7	0	2	1	143945	25088	80554	C:	tempdb	C:\SQL\Log\templog.ldf
8	1	0	1	399983	8192	368640	C:	model	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
9	1	0	1	112867	2673	57770	C:	model	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
10	1	1	1	63301	8192	62595	C:	msdb	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
11	0	0	0	25002	12288	18645	C:	msdb	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
12	0	0	0	22528	8192	19660	C:	tempdb	C:\SQL\Data\tempdb_mssql_2.ndf
13	1	0	0	54515	8271	29044	C:	master	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
14	0	0	0	15797	1405	1465	C:	master	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...
15	0	0	0	59994	8192	54541	C:	tempdb	C:\SQL>Data\tempdb.mdf
16	2	0	0	20211	13178	13191	C:	AdventureWorks	C:\SQL\Log\AdventureWorks2014_Log.ldf
17	0	0	0	61755	8192	60943	C:	Baseline	C:\SQL\Data\baseline_data.mdf

Figure 9-26. IO performance on my test SQL Server instance

One important thing to keep in mind with the IO performance script is that its values are cumulative from the start of the SQL Server service. They do not show the situation at the moment of executing the query. If you are interested in monitoring your IO performance using this script, you can consider capturing the output to a table at a specific interval and calculate the deltas (much in the same way as we did in Chapter 4, “Building a Solid Baseline”).

Next to the performance of your storage subsystem, the behavior of your queries can impact the wait times of the PAGEIOLATCH_[xx] wait type. The more data your queries are requesting (that is not already in the buffer cache), the larger the amount of data that needs to be read from the storage subsystem into the buffer cache. For instance, if we were to modify the query of Listing 9-3, shown in Listing 9-5, to be more selective so that less data is returned, the amount of PAGEIOLATCH_[xx] wait time should also be less.

Listing 9-5. Modified Listing 9-3 query

```
CHECKPOINT 1;
GO
DBCC DROPCLEANBUFFERS;
GO
```

```

DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
GO

SELECT
    SOD.SalesOrderID,
    SOD.CarrierTrackingNumber,
    SOH.CustomerID,
    C.AccountNumber,
    SOH.OrderDate,
    SOH.DueDate
FROM Sales.SalesOrderDetail SOD
INNER JOIN Sales.SalesOrderHeader SOH
ON SOD.SalesOrderID = SOH.SalesOrderID
INNER JOIN Sales.Customer C
ON SOH.CustomerID = C.CustomerID
WHERE SOD.CarrierTrackingNumber BETWEEN 'F467-41BF-8B' AND 'F4E4-4739-B4'
;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'PAGEIOLATCH_%';

```

Figure 9-27 shows the results of the last query that was executed against the sys.
dm_os_wait_stats DMV.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PAGEIOLATCH_NL	0	0	0	0
2	PAGEIOLATCH_KP	0	0	0	0
3	PAGEIOLATCH_SH	5	3	1	0
4	PAGEIOLATCH_UP	0	0	0	0
5	PAGEIOLATCH_EX	0	0	0	0
6	PAGEIOLATCH_DT	0	0	0	0

Figure 9-27. PAGEIOLATCH_SH wait time information

As you can see from Figure 9-27, the wait times of the PAGEIOLATCH_SH wait type went down drastically, from 19 to 3 milliseconds. Now, this example is rather small, and we are dealing with very small result sets, but I think it shows the point.

However, you don't always have the luxury of being able to modify every query so that it is more selective. Maybe the queries are generated by an application and you can't even modify them, or the queries simply need the large result set. Thankfully, as a DBA, we can also play a part in minimizing PAGEIOLATCH_[xx] wait times by simply performing database maintenance. Index fragmentation and out-of-date statistics can increase the PAGEIOLATCH_[xx] wait times drastically. If indexes are fragmented more disk IOs need to take place to retrieve the data requested, which means IO latches will need to stay in place longer, which results in higher PAGEIOLATCH_[xx] wait times. Out-of-date statistics can also result in more disk IOs, because SQL Server expects a different number of rows to be returned instead of the actual number of rows. So, make sure you are regularly performing index and statistics maintenance to make sure the amount of disk interaction is as small as possible.

The final area that can impact PAGEIOLATCH_[xx] wait time is the memory of your system. SQL Server will remove data pages from inside the buffer cache if they have not been accessed within a specific timeframe in order to free up room inside the buffer cache. The interval at which SQL Server performs this cleanup depends on the amount of data coming into the buffer cache and the amount of free space inside the buffer cache. If the request for data pages inside the buffer cache is very high, SQL Server will be forced to swap data pages that have been accessed the least (or haven't been accessed for a while) for pages that are required now. This movement of data pages from and to the buffer cache will result in more PAGEIOLATCH_[xx] waits. In an ideal world, your database would fit completely inside the buffer cache of your SQL Server instance. In this case, SQL Server will only need to move the data pages from the storage subsystem into the buffer cache once, where they will stay until SQL Server restarts again. Even though we do have access to very large amounts of RAM these days, in many cases we cannot simply fit our entire database into the buffer cache of our SQL Server instance, and some swapping of data pages from the buffer cache back to the storage subsystem can be expected. Adding more RAM to your system will increase the number of data pages the buffer cache can store and can help the buffer cache keep those pages in memory longer.

There are two Perfmon counters that can help you get some insight into the buffer cache usage: SQLServer:Buffer Manager\Buffer cache hit ratio and SQLServer:Buffer Manager\Page life expectancy. The SQLServer:Buffer Manager\Buffer cache hit ratio will show you what percentage of pages could be located in the buffer cache that do not require a physical read on the storage subsystem.

The SQLServer:Buffer Manager\Page life expectancy counter will show you the number of seconds a data page stays inside the buffer cache. If you see continuously low values on both these counters, compared to your baseline, it could mean SQL Server is running into memory pressure and needs to move data pages from the buffer cache back to disk again to free up memory. These two counters are not perfect, though, and much has been written about their workings (and specifically their ideal values). We won't go into details about what good values for these counters should be, as for that you should refer to your baseline, but I believe they are a good starting point for investigating buffer cache memory pressure.

PAGEIOLATCH_[xx] Summary

The PAGEIOLATCH_[xx] wait type is, by far, the most common latch-related wait type. Together with the CXPACKET wait type, the PAGEIOLATCH_[xx] wait type is probably the most common wait type on any SQL Server instance. The PAGEIOLATCH_[xx] wait type is directly related to the movement of data pages on the storage subsystem into the buffer cache memory of your SQL Server instance. SQL Server uses the buffer cache to minimize the number of interactions to the (much slower) storage subsystem so as to maximize performance. Whenever a data page is read into the buffer cache, the PAGEIOLATCH_[xx] wait type will be recorded for the time it took to do so. There are many methods available to lower the amount of PAGEIOLATCH_[xx] wait time. The frequently advised "get faster storage" doesn't always hold true, even though fast storage will indeed directly influence the PAGEIOLATCH_[xx] wait times. Optimizing queries so they require fewer data pages to be moved to the buffer cache, performing maintenance on indexes and statistics, and analyzing memory performance could all lead to lower PAGEIOLATCH_[xx] wait times.

CHAPTER 10

High-Availability and Disaster-Recovery Wait Types

There have always been several options available within SQL Server to make sure your database is always available to your users and/or the data inside your database is replicated to another server so as to minimize the chances of losing data. Just like with performing regular database backups to ensure you can revert to a previous state of your database should a crash or data corruption occur, planning and maintaining highly available database environments is part of your job as a DBA.

Now that data has become incredibly important for many companies, and the need for high-availability database servers grows, many DBAs will find themselves managing SQL Server instances inside a high-availability solution, like mirroring, or a disaster-recovery configuration, like log shipping. With these types of SQL Server high-availability and disaster-recovery configurations comes a group of dedicated wait types that are directly related to the health of your high-availability and disaster-recovery (HA/DR) configuration. With the release of SQL Server AlwaysOn Availability Groups in SQL Server 2012, more options became available for configuring HA/DR solutions, together with new wait types that are directly related to AlwaysOn Availability Groups.

In this chapter we will take a look at some of the most common wait types to see in HA/DR configurations. The main focus of the wait types inside this chapter is AlwaysOn Availability Groups, because Microsoft is deprecating many of the previous installments of features that now fall under the name AlwaysOn Availability Groups, like mirroring. As an exception to this rule, I selected one mirroring-related wait type that is relatively common on highly used mirroring configurations. All the other wait types are related to AlwaysOn Availability Groups.

For the examples inside this chapter, I used several virtual machines to create a mirroring and an AlwaysOn Availability Groups configuration. The configuration of these VMs can be found in Appendix I Example SQL Server Machine Configurations.

DBMIRROR_SEND

The first wait type I want to discuss in this chapter is the DBMIRROR_SEND wait type. As you might suspect from the wait type name, DBMIRROR_SEND is related to database mirroring.

Database mirroring is a feature that was introduced in SQL Server 2005 but was announced deprecated in SQL Server 2012. This doesn't mean you cannot use database mirroring in SQL Server 2012 or SQL Server 2014, but it does mean it is scheduled for removal. The entire feature will be replaced with AlwaysOn Availability Groups, which offers the same configuration options as database mirroring.

Database mirroring is a solution that increases the availability of SQL Server databases, and unlike, for instance, failover clustering, it can be configured on a per-database basis. Database mirroring works by redoing every data modification operation that occurs on the primary database (called *principal* in database-mirroring terms) on the mirror database. The redoing of every database modification operation is achieved by streaming active transaction log records to the mirror server, which will perform the operations on the mirror database in the sequence in which they were inserted into the transaction log on the principal database.

Database mirroring offers two different operating modes that impact the availability and performance of the mirror configuration: synchronous (or high-safety) mode and asynchronous (or high-performance) mode. Even though both modes perform identical actions to ensure data modification operations are also performed on the mirror database, there can be a large difference in performance, and thus in waits occurring.

The synchronous mirror mode makes sure that every data modification action that is performed on the principal is also directly performed on the mirror. It does this by waiting on sending a transaction confirmation message to the client until the transaction is successfully written to disk on the mirror. Figure 10-1 depicts synchronous mirroring.

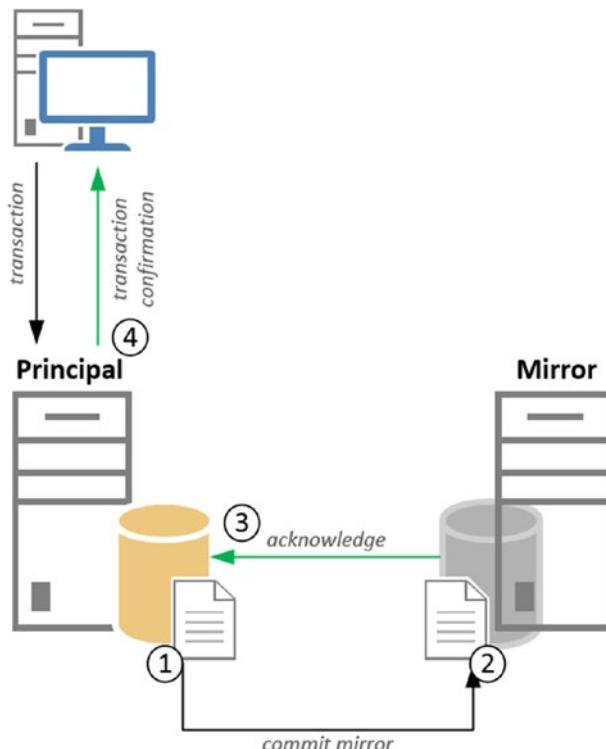


Figure 10-1. Synchronous mirroring

Even though synchronous mirroring makes sure data on the principal and mirror are 100% identical, it comes with a few drawbacks. One of those is that the performance of your database inside a synchronous mirroring configuration is highly dependent on the speed the mirror can process data modification operations, since every transaction has to be committed on the mirror first.

The flow of a data modification transaction is described in the steps that follow:

1. When the transaction is received, the principal will write the transaction to the transaction log, but the transaction is not yet committed though.
2. The principal will send the log record to the mirror.
3. The mirror will harden the log record to disk and send an acknowledgment to the principal.

4. After the principal receives the acknowledgment, it will send a confirmation message to the client that the transaction was completed, and the transaction gets committed to the transaction log on the principal.

The asynchronous mode works in much the same way; the exception is that it will not wait on an acknowledgment message from the mirror before sending the transaction confirmation message to the client. This means that transactions are committed to disk on the principal before they are written to disk on the mirror. Using asynchronous mirroring will improve mirror performance, since the latency overhead of synchronous mirroring is removed. The trade-off for this increase in performance is that asynchronous replication can lead to data loss in the case of a disaster, since it is possible that transactions were not yet committed on the mirror. Figure 10-2 shows the transaction-log flow on an asynchronous mirror; the dotted lines indicate that the actions are not performed directly.

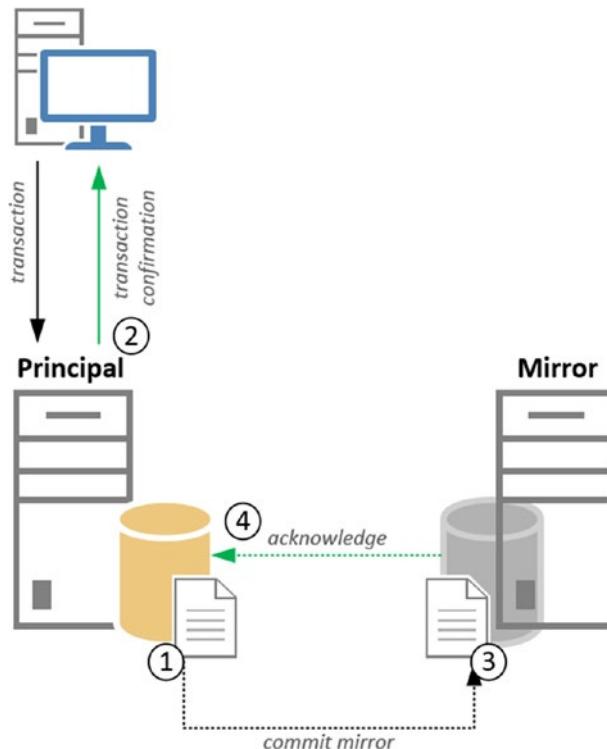


Figure 10-2. Asynchronous mirroring

What Is the DBMIRROR_SEND Wait Type?

The DBMIRROR_SEND wait type is most frequently related to synchronous mirroring configurations. The description of the DBMIRROR_SEND wait type on Books Online is “Occurs when a task is waiting for a communications backlog at the network layer to clear to be able to send messages. Indicates that the communications layer is starting to become overloaded and affect the database mirroring data throughput.” In this case the Books Online description is pretty accurate, but the network is not the only thing that can impact DBMIRROR_SEND wait times. Having a slow disk subsystem connected to the mirror database can, for instance, also lead to an increase in DBMIRROR_SEND wait times.

Another important point to remember is that high DBMIRROR_SEND wait times will frequently only be recorded on the mirror instance, and not on the principal. It is common to see waits occur on the DBMIRROR_SEND wait type on both the principal and the mirror, but these will normally be very low on the principal. They can still reach high values on the mirror since, generally, there is always some latency between both SQL Server instances. Because of expected latency, I advise you to use baseline measurements to identify higher-than-normal wait times for the DBMIRROR_SEND wait type.

DBMIRROR_SEND Example

For this example I have built a synchronous mirror between two of my test SQL Server instances, using the AdventureWorks database as the database that will be mirrored between both instances.

Inside the AdventureWorks database, I create a simple table using the script in Listing 10-1.

Listing 10-1. Create Mirror_Test table

```
USE [AdventureWorks]
GO

CREATE TABLE Mirror_Test
(
    ID UNIQUEIDENTIFIER PRIMARY KEY,
    RandomData VARCHAR(50)
);
```

After the table is created, I clear the sys.dm_os_wait_stats DMV and insert 10,000 rows into the Mirror_Test table using the query in Listing 10-2. I also make sure to clear the sys.dm_os_wait_stats DMV on the mirror as well before running the script in Listing 10-2.

Listing 10-2. Insert 10,000 rows into Mirror_Test table

```
DBCC SQLPERF('sys.dm_os_wait_stats, CLEAR')

INSERT INTO Mirror_Test
(
ID,
RandomData
)
VALUES
(
NEWID(),
CONVERT(VARCHAR(50), NEWID())
);
GO 10000
```

While the script is running, I look at the DBMIRROR_SEND wait times on both the mirror and the principal using the following query:

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'DBMIRROR_SEND'
```

The results of this query can be seen in Figure 10-3, which shows the DBMIRROR_SEND wait times on the mirror. Figure 10-4 shows the DBMIRROR_SEND wait times on the principal server.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	DBMIRROR_SEND	19146	23030	3452	900

Figure 10-3. DBMIRROR_SEND wait times on the mirror

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	DBMIRROR_SEND	0	0	0	0

Figure 10-4. DBMIRROR_SEND wait times on the principal

As you can see, we spend quite some time waiting on the DBMIRROR_SEND wait type on the mirror vs. no DBMIRROR_SEND waits on the principal.

Lowering DBMIRROR_SEND Waits

One of the most common pieces of advice for lowering DBMIRROR_SEND wait time is changing the mirror mode from synchronous to asynchronous. While this will absolutely lower the wait time, it also means you can potentially lose data when a disaster occurs on the principal. Lowering the wait time on the DBMIRROR_SEND wait type will have a positive effect on the duration of your queries. For instance, in the example in the previous section, the insert of 10,000 rows took around 30 seconds on my test SQL Server mirror configuration. When I changed the mirror mode from synchronous to asynchronous, not only did the wait times on the DBMIRROR_SEND wait type go down, the total execution time of 10,000 inserts went down to 3 seconds. That's an improvement of almost 30 seconds!

Even though these improvements might sound very attractive, sometimes changing the mirror mode is not an option. For instance, your company's disaster-recovery strategy can require a synchronous mirror configuration. Changing the mirror mode from synchronous to asynchronous should, in my opinion, be the last option (if it actually is a viable option). There are other parts that can influence DBMIRROR_SEND wait times, like the storage configuration on the mirror or the network connection between the principal and mirror SQL Server instances. Both these parts can act like a bottleneck between both instances, contributing to the DBMIRROR_SEND wait time.

Next to checking out the performance of your storage subsystem and network connection, SQL Server has a database mirroring monitor that will give you status information about the mirroring configuration. You can find the database mirroring monitor by right-clicking the database that is part of a mirror, selecting Tasks ► Database Mirroring Monitor. Figure 10-5 shows the monitor against my test mirror configuration.

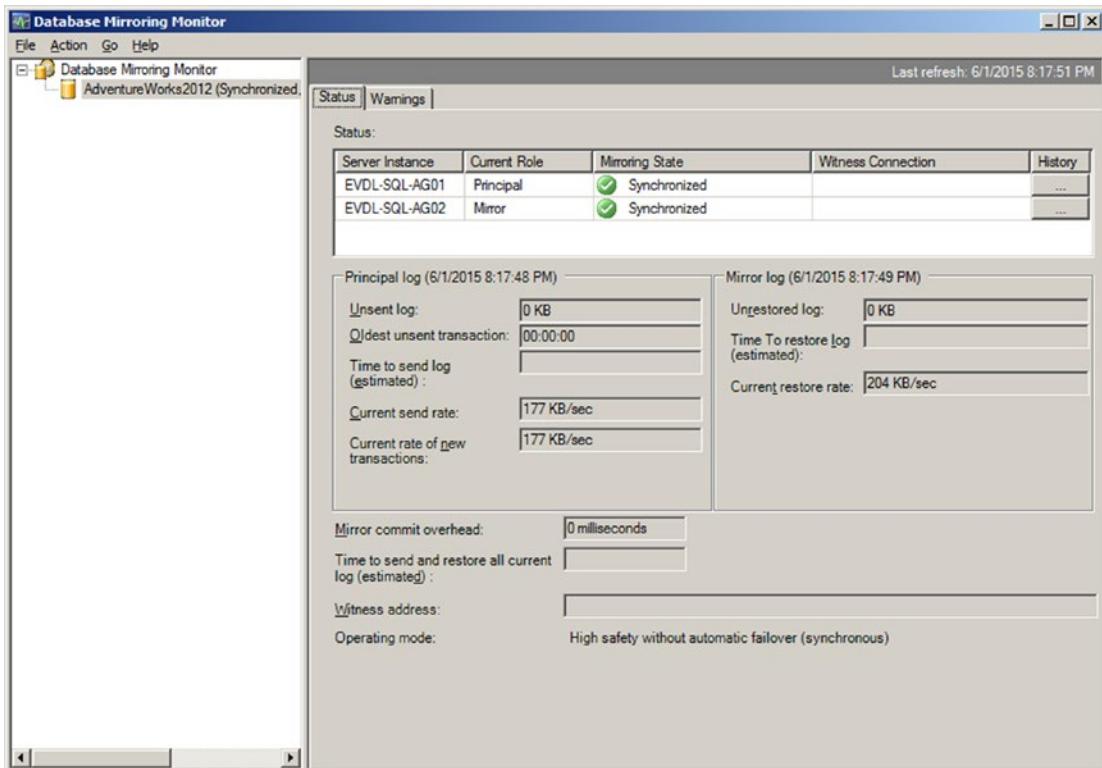


Figure 10-5. Database mirroring monitor

As you can see, the database mirroring monitor can provide you with some very interesting additional information like the number of log records that still need to be sent or restored, how far behind the mirror currently is, and the send and restore rates. In many of my dealings with database mirroring, the database mirroring monitor is the first place I'll check when there are performance issues involving the mirror configuration.

DBMIRROR_SEND Summary

The DBMIRROR_SEND wait type is directly related to database mirroring. Seeing DBMIRROR_SEND waits occur is pretty normal on most mirror configurations. This makes using a baseline to identify wait time spikes a necessity. The mirroring mode plays a huge part in the DBMIRROR_SEND wait times. When using synchronous mirroring, DBMIRROR_SEND wait times will frequently be higher than when using asynchronous mirroring. Not only

the mirroring mode influences DBMIRROR_SEND wait times, though. Having a storage subsystem on the mirror SQL Server instance that cannot keep up with the load will have an effect on DBMIRROR_SEND waits, just like the network connection between the principal and the mirror.

HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE

The HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types are both related to AlwaysOn Availability Groups. All wait types that are related to AlwaysOn can easily be identified by the HADR_ prefix in the wait type's name. AlwaysOn Availability Groups was introduced in SQL Server 2012 as a replacement for various SQL Server high-availability and disaster-recovery features such as database mirroring. There are quite a few different wait types associated with AlwaysOn, totaling 65 in SQL Server 2017. Not all of these wait types necessarily indicate performance problems somewhere in your AlwaysOn configuration. The HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE are both perfect examples of benign wait types that occur naturally over time and do not directly indicate a performance problem. Since both these wait types have high wait times associated with them on every AlwaysOn configuration, and are thus very common, I wanted to include them in this chapter to help you better understand what function they have.

What Are the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE Wait Types?

As I mentioned in the preceding section, both the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types occur in AlwaysOn configurations. They both occur in different places inside your AlwaysOn configuration and have slightly different functions.

According to Books Online, the HADR_LOGCAPTURE_WAIT wait type indicates that SQL Server is “waiting for log records to become available. Can occur either when waiting for new log records to be generated by connections or for I/O completion when reading log not in the cache. This is an expected wait if the log scan is caught up to the end of log or is reading from disk.” The HADR_LOGCAPTURE_WAIT wait type occurs on the SQL Server that hosts the primary database inside an AlwaysOn Availability Group. Think of the primary database as being just like the principal inside a database mirroring configuration.

AlwaysOn works much the same way as database mirroring and also provides two different modes (called Availability modes inside AlwaysOn): Synchronous-commit and Asynchronous-commit. Both these Availability modes work in the same way as their database mirroring counterparts we were discussing earlier in this chapter do. This means that in Synchronous-commit mode the primary replica waits to commit transactions to the transaction log until the secondary replica has completed its own log hardening, while in Asynchronous-commit mode the primary replica will directly commit the transaction to the transaction log without waiting for a confirmation from the secondary replica.

While the primary replica is waiting for work, SQL Server will record the time it has spent on waiting for new transactions to become available as the HADR_LOGCAPTURE_WAIT wait type. This means that seeing high wait times on the HADR_LOGCAPTURE_WAIT wait type actually means that SQL Server is waiting on new transactions to become available so they can be transferred to the secondary replica. This is not dependent on the Availability mode you configured for your AlwaysOn Availability Group. The HADR_LOGCAPTURE_WAIT wait type will always occur, no matter your AlwaysOn configuration. Figure 10-6 shows an AlwaysOn Availability Group configuration together with the HADR_LOGCAPTURE_WAIT wait type on the primary replica, which occurs while waiting for new transactions to be sent to the secondary replica.

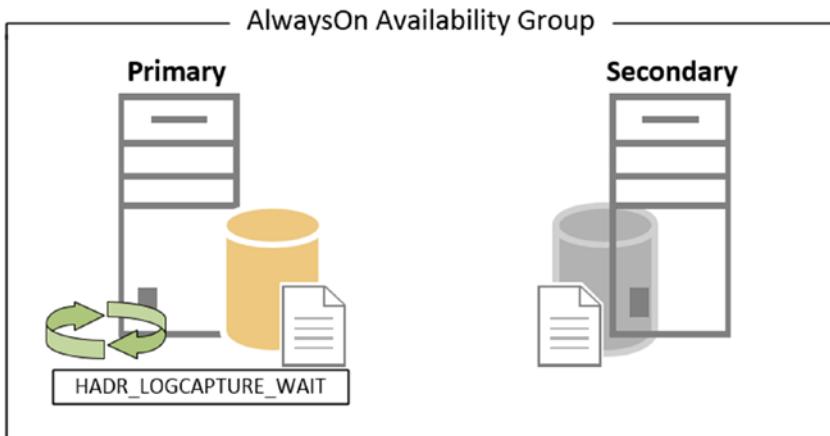


Figure 10-6. AlwaysOn Availability Group and the HADR_LOGCAPTURE_WAIT wait type

Even though I placed the `HADR_LOGCAPTURE_WAIT` wait type on the primary replica in Figure 10-6, it will also log the `HADR_LOGCAPTURE_WAIT` wait type on the secondary replica, although those values will normally be much lower than on the primary replica.

The `HADR_WORK_QUEUE` wait type is almost identical in function to the `HADR_LOGCAPTURE_WAIT` wait type. Books Online gives an excellent description of this wait type: “AlwaysOn Availability Groups’ background worker thread waiting for new work to be assigned. This is an expected wait when there are ready workers waiting for new work, which is the normal state.” The main difference between both wait types, is that the `HADR_LOGCAPTURE_WAIT` wait type is dedicated to waiting until new transactions become available, while the `HADR_WORK_QUEUE` indicates that are free threads waiting for work. Just like the `HADR_LOGCAPTURE_WAIT` wait type, the `HADR_WORK_QUEUE` occurs on both the primary and the secondary replicas, but the `HADR_WORK_QUEUE` wait type is much more prevalent on both replicas. As a matter of fact, the `HADR_WORK_QUEUE` wait type will frequently be the top AlwaysOn related wait type on every SQL Server that is part of an AlwaysOn Availability Group, especially if the work load is low.

Figure 10-7 shows an AlwaysOn Availability Group like the one in Figure 10-6, but this time I added the `HADR_WORK_QUEUE` wait type to the image as well.

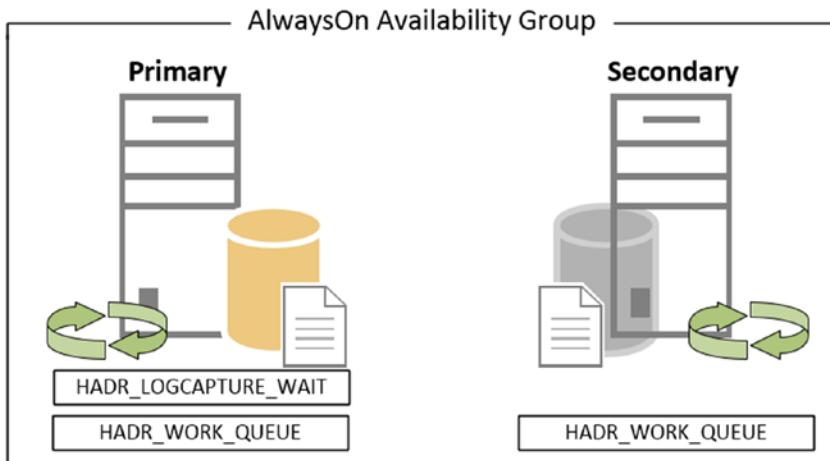


Figure 10-7. *HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types*

Since both the `HADR_LOGCAPTURE_WAIT` and `HADR_WORK_QUEUE` wait types occur naturally over time, I did not include an example of both the wait types. Also, because both these wait types are not directly related to performance problems, there is no use including a section on lowering the wait times of both these wait types.

HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE Summary

Both the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types are benign wait types that occur on every SQL Server that is part of an AlwaysOn Availability Group. Because the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types are not directly related to performance problems, there is no direct need to focus attention on lowering them, and they can, in most cases, be safely ignored.

HADR_SYNC_COMMIT

The HADR_SYNC_COMMIT wait type is another AlwaysOn-related wait type that was introduced in SQL Server 2012. In many ways the HADR_SYNC_COMMIT wait type closely resembles the DBMIRROR_SEND wait type we discussed earlier in this chapter. There are some differences, however, between both wait types, which we will discuss in the following section.

What Is the HADR_SYNC_COMMIT Wait Type?

The HADR_SYNC_COMMIT wait type indicates the time the primary replica spends waiting for the secondary replica to harden the log records. HADR_SYNC_COMMIT waits will only occur on the primary replica and only inside a synchronous-replication AlwaysOn Availability Group. As soon as a transaction is received by the primary replica and is sent to the secondary replica for hardening, the HADR_SYNC_COMMIT wait time will start recording. The HADR_SYNC_COMMIT wait time will only stop recording when the secondary replica has sent its confirmation that the write to the secondary's transaction log was completed. Figure 10-8 shows the HADR_SYNC_COMMIT wait time generation inside a timeline.

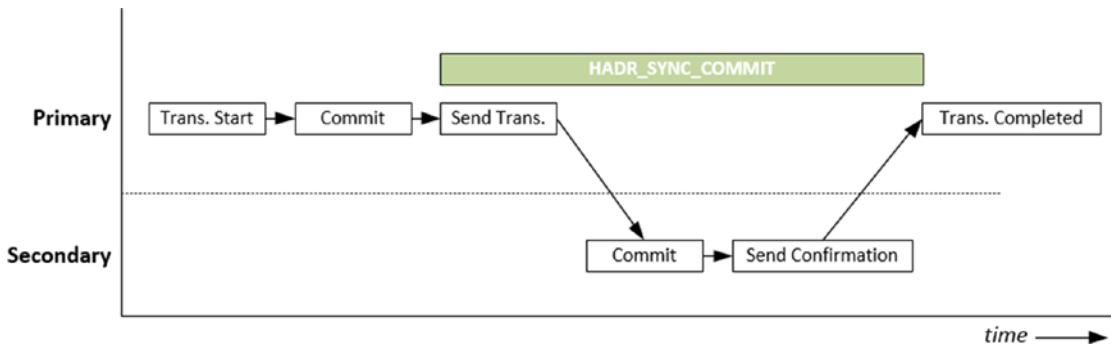


Figure 10-8. *HADR_SYNC_COMMIT and synchronous replication*

Since the HADR_SYNC_COMMIT wait type will always occur in every synchronous replicated AlwaysOn Availability Group, it is normal to expect a certain amount of wait time. But just like the DBMIRROR_SEND wait type, the wait time of the HADR_SYNC_COMMIT wait type is highly dependent on the speed at which the secondary replica can process the log records. This means that a slow network connection between both replicas or the performance of the storage subsystem on the secondary replica can impact HADR_SYNC_COMMIT wait times. For this reason, it is important to understand what the normal wait times for the HADR_SYNC_COMMIT wait type are for your AlwaysOn configuration so you can identify higher-than-normal wait times easily.

HADR_SYNC_COMMIT Example

For this example, I have built an AlwaysOn Availability Group configured to use synchronous replication. The configuration of the test machines I will use for this can be found in Appendix I Example SQL Server Machine Configuration. I won't go into detail on how you can configure an AlwaysOn Availability Group, as there is plenty of information available on the Internet to help you configure AlwaysOn. A good starting point is the "Getting Started with AlwaysOn Availability Groups" article on Books Online, which you can find here: <https://msdn.microsoft.com/en-us/gg509118>. I used the AdventureWorks database as the database that needed to be replicated inside my AlwaysOn Availability Group.

After my AlwaysOn Availability Group was configured I added an extra table named A0_Test to the AdventureWorks database using the script in Listing 10-3.

Listing 10-3. Create AO_Test table

```
USE [AdventureWorks]
GO

CREATE TABLE AO_Test
(
    ID UNIQUEIDENTIFIER PRIMARY KEY,
    RandomData VARCHAR(50)
);
```

After the table is created, I first clear and then query the sys.dm_os_wait_stats DMV to check the current wait times on the HADR_SYNC_COMMIT wait time on both the primary and secondary replicas using the following query:

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'HADR_SYNC_COMMIT';
```

Even after waiting for a couple of minutes, the wait time of the HADR_SYNC_COMMIT wait type stays 0, as you can see in Figure 10-9. This is what I expected since we have not performed any data modifications on the primary replica so far.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	HADR_SYNC_COMMIT	0	0	0	0

Figure 10-9. HADR_SYNC_COMMIT wait information during no activity on both the primary and the secondary mode

This is different compared to HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE, which will accumulate wait times even though (or because) there is no user activity inside the AlwaysOn Availability Group.

Now that the table is in place, let's generate some transactions by performing a number of inserts. The script in Listing 10-4 will insert 10,000 rows into the AO_Test table we created earlier.

Listing 10-4. Insert 10,000 rows into the AO_Test table

```
INSERT INTO AO_Test
(
ID,
RandomData
)
VALUES
(
NEWID(),
CONVERT(VARCHAR(50), NEWID())
);
GO 10000
```

When the script in Listing 10-4 has completed, I check the wait statistics information inside the sys.dm_os_wait_stats DMV again on both the primary and secondary replicas. Figure 10-10 shows the results of this query on the primary, and Figure 10-11 on the secondary.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	HADR_SYNC_COMMIT	10000	15342	41	211

Figure 10-10. HADR_SYNC_COMMIT waits on the primary replica

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	HADR_SYNC_COMMIT	0	0	0	0

Figure 10-11. HADR_SYNC_COMMIT waits on the secondary replica

The first thing you will notice when looking at both figures is that the HADR_SYNC_COMMIT waits only occur on the primary replica and not on the secondary, which is expected behavior. The second interesting thing is the number of waits that occurred. This is the exact same amount as the number of rows we inserted. Again, this is expected behavior. Since we performed a single insert and just repeated it 10,000 times, every insert generated a single transaction-log record that needed to be replicated. Using the number of waits that occurred and the wait time, it is possible to calculate the average time it took for one insert operation to be committed on the replica. In this case it is 1.53 milliseconds ($15342/10000$) which is a pretty decent value.

Lowering HADR_SYNC_COMMIT Waits

Seeing HADR_SYNC_COMMIT waits occur does not necessarily mean there is a problem. HADR_SYNC_COMMIT waits will always occur whenever there are data modifications performed on your primary replica. They can indicate a problem if the wait times are much higher than you expect them to be when you compare them to your baseline measurements.

Changing the AlwaysOn operation mode to asynchronous replication will completely remove HADR_SYNC_COMMIT waits, but at the risk of losing data when a disaster occurs. Also, to reach your company's disaster-recovery or high-availability needs, you frequently do not have the luxury of just changing the AlwaysOn operating mode, and I advise you not to change it just to lower HADR_SYNC_COMMIT wait times.

Thankfully, there are many different methods you can use to monitor the performance of your AlwaysOn Availability Group, including the AlwaysOn Dashboard, DMVs, and Perfmon counters.

You can open the AlwaysOn Dashboard by right-clicking your AlwaysOn Availability Group and selecting the "Show Dashboard" option. The AlwaysOn Dashboard, by default, gives you some general information, like the servers inside the Availability Group and the synchronization state, about your AlwaysOn Availability Group, as shown in Figure 10-12.

The screenshot shows the AlwaysOn Dashboard for the EVDL-AG01 Availability Group. At the top, a green checkmark icon indicates the group is healthy. Below it, the group state is listed as "Healthy". The primary instance is "EVDL-SQL-AG01" and the failover mode is "Manual". The cluster state is "EVDL-SQL-AG (Normal Quorum)".

Availability replica:

Name	Role	Failover Mode	Synchronization State	Issues
EVDL-SQL-AG01	Primary	Manual	Synchronized	
EVDL-SQL-AG02	Second...	Manual	Synchronized	

Group by

Name	Replica	Synchronization State	Failover Readin...	Issues
EVDL-SQL-AG01				
AdventureWorks2012	EVDL-SQL-AG01	Synchronized	No Data Loss	
EVDL-SQL-AG02				
AdventureWorks2012	EVDL-SQL-AG02	Synchronized	No Data Loss	

Figure 10-12. AlwaysOn Dashboard

The default view of the AlwaysOn Dashboard doesn't provide much information you can use for troubleshooting. Thankfully, you can configure the view to suit your own needs by right-clicking the column bar and selecting the information you are interested in, as shown in Figure 10-13.

The screenshot shows the AlwaysOn Dashboard for the availability group EVDL-AG01. At the top, it displays the availability group state as healthy, primary instance as EVDL-SQL-AG01, failover mode as manual, and cluster state as EVDL-SQL-AG (Normal Quorum). Below this, there's a section for 'Availability replica:' showing two replicas: EVDL-SQL-AG01 (Primary, Manual) and EVDL-SQL-AG02 (Secondary, Manual). A context menu is open over the 'Name' column header of the replica table, listing various performance and status metrics. The 'Log Send Queue Size (KB)' option is highlighted with a blue selection bar.

Name	Role	Failover Mode
EVDL-SQL-AG01	Primary	Manual
EVDL-SQL-AG02	Second...	Manual

Name	Replica
EVDL-SQL-AG01	
AdventureWorks2012	EVDL-SQL-AG01
EVDL-SQL-AG02	
AdventureWorks2012	EVDL-SQL-AG02

- Name
- Replica
- Synchronization State
- Failover Readiness
- Issues
- Suspended
- Suspend Reason
- Estimated Recovery Time (seconds)
- Estimated Data Loss (time)
- Synchronization Performance (seconds)
- Log Send Queue Size (KB)
- Log Send Rate (KB/sec)
- Redo Queue Size (KB)
- Redo Rate (KB/sec)

Figure 10-13. AlwaysOn add columns

There are many columns that are interesting for troubleshooting synchronization issues, and I recommend taking the time to understand them so you can determine which columns are most applicable to your situation.

The information shown by the AlwaysOn Dashboard is originally recorded inside various AlwaysOn-related DMVs. This makes it possible for you to query this information yourself. All of the AlwaysOn-related DMVs can easily be identified by the `dm_hadr` prefix in the DMV name, like the `sys.dm_hadr_database_replica_states` DMV that contains a large part of the information you can access inside the AlwaysOn Dashboard.

Next to the AlwaysOn and DMVs that are related to AlwaysOn, there are a large amount of Perfmon counters that specifically show AlwaysOn performance. These counters are grouped in the Perfmon SQLServer:Availability Replica and SQLServer:Database Replica groups. Figure 10-14 shows a part of the counters available in the SQLServer:Database Replica group.

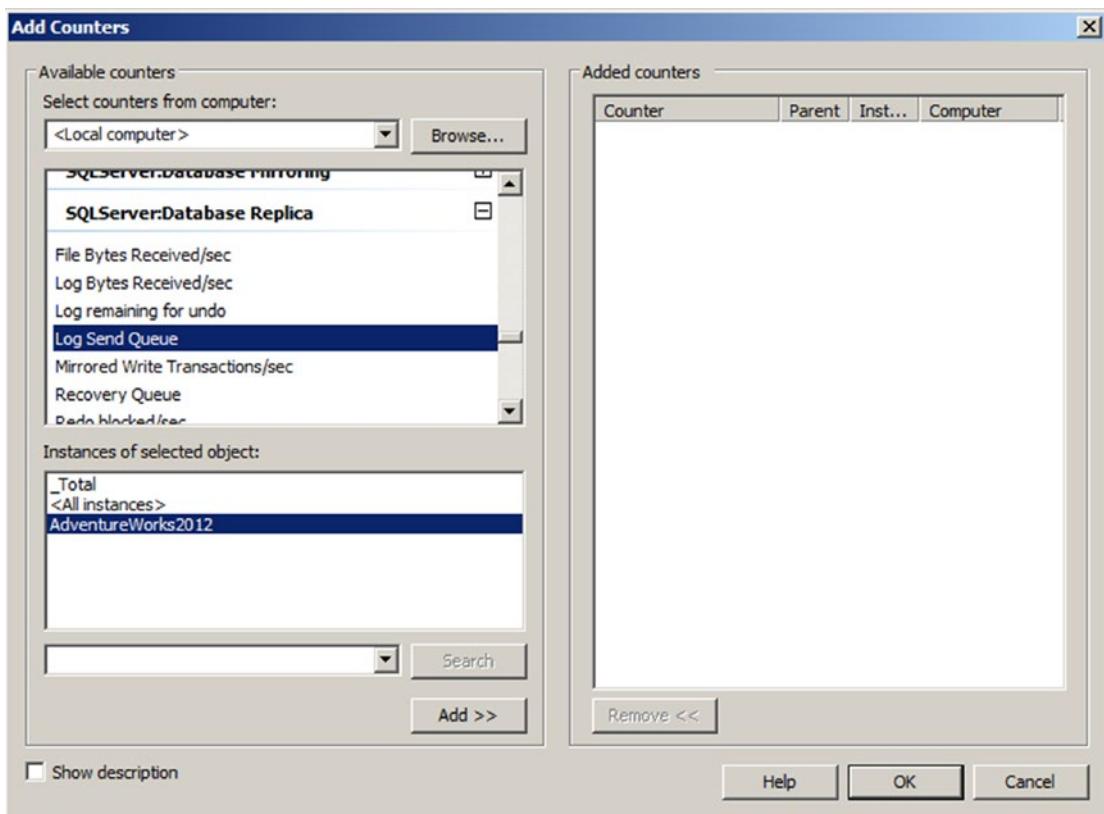


Figure 10-14. Perfmon counters related to AlwaysOn

As you have read so far, there are plenty of options available to you for analyzing the AlwaysOn performance between replicas.

Using the information from the various sources I have shown you so far, you should be able to check the general health of your AlwaysOn Availability Group. You can then combine this information with other metrics for things that impact the performance of your secondary replica, like the performance of your storage subsystem and your network connection. Since the HADR_SYNC_COMMIT wait type is strictly related to the secondary replica, you should focus your analysis on the SQL Server that hosts the

secondary replica. For instance, if your storage subsystem cannot keep up with the number of transactions that need to be committed on the secondary replica, you will notice this in higher HADR_SYNC_COMMIT wait times, and also in the various counters inside the AlwaysOn Dashboard, DMVs, or Perfmon.

It is difficult to give a general recommendation on how to lower HADR_SYNC_COMMIT wait times since they are highly dependent on myriad variables and also depend on your workload. When you have a workload that consists of a large number of read queries, you will notice lower HADR_SYNC_COMMIT wait times than workloads that perform many data modification operations. This means analyzing and optimizing your query workload can also contribute to the lowering of HADR_SYNC_COMMIT wait times.

HADR_SYNC_COMMIT Summary

The HADR_SYNC_COMMIT wait type will only occur on AlwaysOn Availability Groups that consist of replicas that are configured to use the synchronous replication mode. The HADR_SYNC_COMMIT wait type will give you insight into how long it took for the secondary replica to commit the transaction to disk. Since the HADR_SYNC_COMMIT will always record wait times inside synchronous replication, you should only worry about the wait times when they are far higher than expected. Thankfully, there are various methods available to you to analyze the performance of your AlwaysOn Availability Group, including an AlwaysOn Dashboard, DMVs, and Perfmon counters.

Since the performance of the secondary replica has the largest impact on the HADR_SYNC_COMMIT wait times, your attention should focus on the secondary replica when troubleshooting this wait type. The storage subsystem and network connection both play a large role in the speed at which the secondary replica can write log records to its transaction log. Your workload also impacts HADR_SYNC_COMMIT wait times, and optimizing it so data modifications are better spread out will result in lower HADR_SYNC_COMMIT wait times.

REDO_THREAD_PENDING_WORK

The last wait type in this chapter is the REDO_THREAD_PENDING_WORK wait type. And even though it misses the characteristic HADR_ prefix that identifies AlwaysOn-related wait types, it is related to AlwaysOn. The REDO_THREAD_PENDING_WORK wait type is, just like the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types, a wait type that accumulates

over time when there is no work to be done. And just like the HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE wait types, it can in most cases be ignored since it does not indicate a performance problem.

Even though this is a wait type that can safely be ignored in 99% of the cases, I wanted to include it in this chapter for two reasons. It is usually one of the top wait types on an AlwaysOn Availability Group secondary replica, and understanding its related process inside SQL Server will give you a better understanding of the inner workings of AlwaysOn.

What Is the REDO_THREAD_PENDING_WORK Wait Type?

The REDO_THREAD_PENDING_WORK wait type is related to a process that only occurs on the secondary replica inside an AlwaysOn Availability Group, the Redo Thread.

Up to this point in the chapter, we have talked about how the secondary replica inside an AlwaysOn Availability Group processes log records, hardens them to its own transaction log, and sends a confirmation to the primary replica. When using synchronous replication, the primary replica will wait before sending a transaction complete message to the client that started the transaction, and when using asynchronous replication the message is sent without waiting for the hardening on the secondary. But until now we haven't discussed the process that will perform the modifications inside the secondary database described in the log records. This is where the Redo Thread on the secondary comes in. This thread is responsible for performing the data modifications that were recorded in the log records the primary replica sent it. There is one very important concept associated with the Redo Thread: it does not impact the commit confirmation from the secondary replica. This means that the Redo Thread might be performing work long after the transaction has been communicated as committed to the client (both the primary and secondary replica have hardened the log record and the AlwaysOn Availability Group has the synchronized status).

This means that even though your AlwaysOn Availability Group is synchronized, the data inside the secondary database does not necessarily have to be identical to the primary database. This actually matters less than you might think on first thought. Because the secondary hardened the log records to its own transaction log on disk, it has all the information it needs to perform the redo operation. Transactions will not be lost if a failure occurs on the primary since the secondary has all the transactions that were performed in its own transaction log and can redo all the transactions. This works much

the same as a standalone SQL Server instance where transactions are also hardened to disk first before data is actually changed. If SQL Server were to crash in this situation, SQL Server would use the transaction log to redo or undo the data modifications. Figure 10-15 shows an example of synchronous replication together with the Redo Thread. Note that the Redo Thread is a separate operation that does not impact the duration of the transaction complete message.

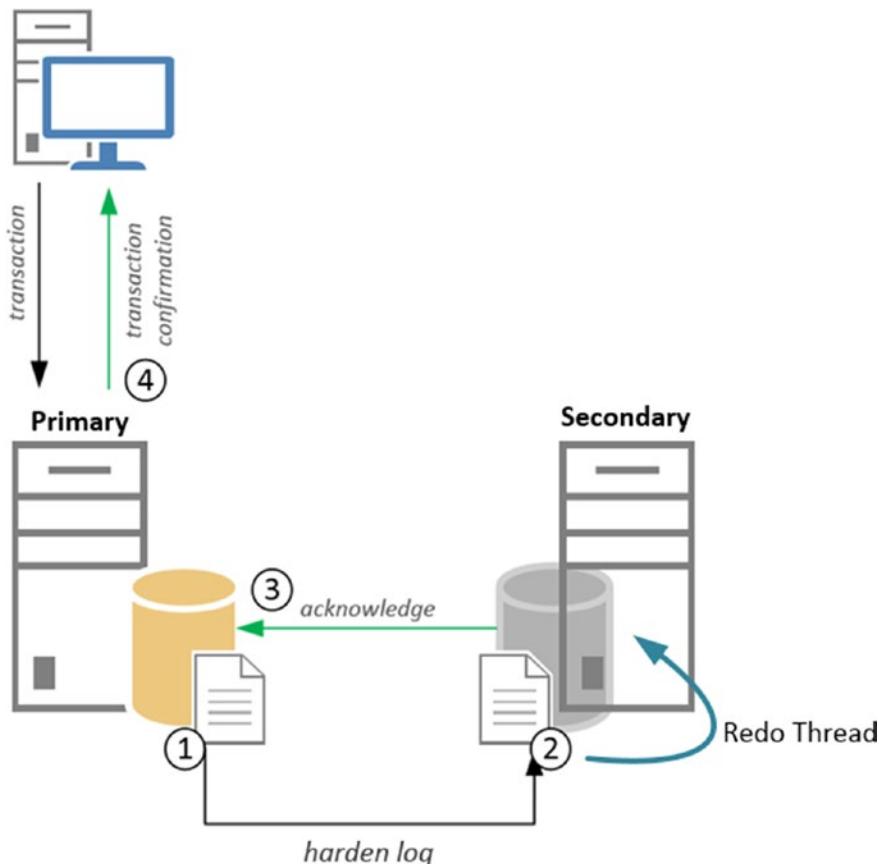


Figure 10-15. Synchronous AlwaysOn Availability Group and the Redo Thread

So, where does the REDO_THREAD_PENDING_WORK wait type come in? Well, if the Redo Thread is waiting for work to arrive, it will record the time it is inactive as wait time on the REDO_THREAD_PENDING_WORK wait type. This will occur on both synchronous and asynchronous replication modes, but only on the secondary replica.

Because the wait type only indicates that the Redo Thread is not performing any work, it can, save for extremely rare cases, be safely ignored. And because the wait time for the REDO_THREAD_PENDING_WORK wait type will accumulate naturally when there is no work to be done, there is no need to write an example demonstrating the wait type. A simple query to retrieve REDO_THREAD_PENDING_WORK wait type information against the sys.dm_os_wait_stats DMV on a secondary replica will show you that the wait time increases, especially when there is no user activity against the AlwaysOn Availability Group, as shown in Figure 10-16.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	REDO_THREAD_PENDING_WORK	143546	14675243	54	11004

Figure 10-16. REDO_THREAD_PENDING_WORK wait information

REDO_THREAD_PENDING_WORK Summary

The REDO_THREAD_PENDING_WORK wait type is an AlwaysOn-related wait type that accumulates wait time naturally over time when there is no data modification activity against an AlwaysOn Availability Group. The REDO_THREAD_PENDING_WORK wait type is related to the Redo Thread on the secondary replica inside an AlwaysOn Availability Group, and it indicates that the Redo Thread is currently waiting for work. Since this wait type will occur on every secondary replica, especially when there is minimal to no user data modification occurring, it can safely be ignored.

CHAPTER 11

Preemptive Wait Types

In Chapter 1, “Wait Statistics Internals,” we briefly touched upon SQL Server’s non-preemptive scheduling model that is used to perform thread scheduling and management. Unlike SQL Server, the Windows operating system uses preemptive scheduling to schedule and manage threads. Sometimes SQL Server has to use Windows functions to perform specific actions through the operating system, for instance, when checking Active Directory permissions. When this occurs, SQL Server will have to ask a thread from the Windows operating system, outside of SQL Server, thus making it impossible for SQL Server to manage that thread. While SQL Server is waiting for the preemptive thread inside the Windows operating system to complete, SQL Server will record a wait on a preemptive wait type. Figure 11-1 shows a graphical representation of this behavior.

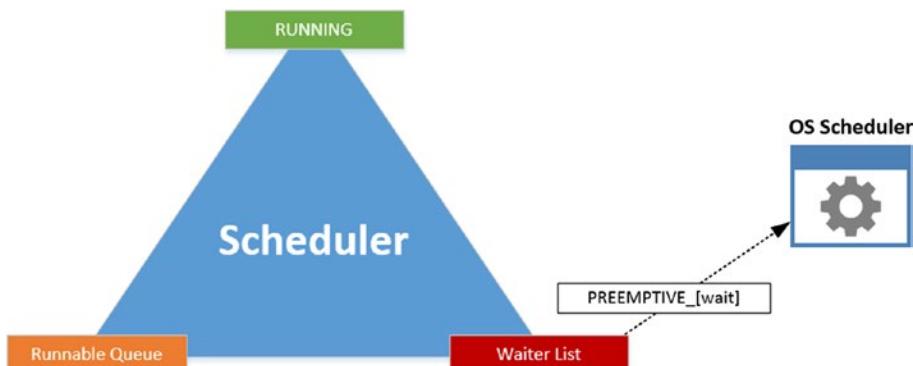


Figure 11-1. Preemptive wait occurring

There are many different preemptive wait types inside SQL Server; at the time of writing this book SQL Server 2017 has 203 different preemptive wait types. Which preemptive wait type is recorded when a thread is requested outside SQL Server depends on the Windows function the thread is accessing. Each of the preemptive

wait types inside SQL Server represents a different Windows function (save for some exceptions that act as a catch-all wait type for different functions), and in many cases the name of the wait type is identical to the name of the Windows function. This is very helpful because you can search for the specific Windows function on MSDN and learn what the function does. If you know what the function does, you also know why, or on what, SQL Server is waiting. For example, if you notice high wait times on the PREEMPTIVE_OS_WRITEFILEGATHER wait type, you can remove the PREEMPTIVE_OS_ part and search MSDN for the WRITEFILEGATHER function. Figure 11-2 shows the results I got on my search for the WRITEFILEGATHER function.

WriteFileGather function

Retrieves data from an array of buffers and writes the data to a file.

The function starts writing data to the file at a position that is specified by an [OVERLAPPED](#) structure. The **WriteFileGather** function operates asynchronously.

Figure 11-2. *WriteFileGather Windows function*

By reading the article we can learn a lot about this function; apparently this function is used when writing data to a file and has to occur outside SQL Server. I won't spoil anything else here, since we will go into more detail about the PREEMPTIVE_OS_WRITEFILEGATHER wait type a bit further down in this chapter.

I won't describe every possible preemptive wait type in this chapter, since there are simply too many of them. Instead I have focused on the most common preemptive wait types. If you run into a preemptive wait type that is not discussed in detail in this chapter, I suggest you use the preceding method to find more information about the Windows function on MSDN. Hopefully, that information can help you figure out why the wait is occurring.

SQL Server on Linux

In the introduction of this chapter, I wrote about how SQL Server can access Windows operating system functionality from inside SQL Server. However, starting from SQL Server 2017, SQL Server is no longer limited to being available on the Microsoft Windows

operating system. In a revolutionary announcement in March of 2016, Microsoft announced the next release of SQL Server (2017) will no longer be a Windows-only product, but will also be available on Linux. Needless to say, the announcement stirred up quite a bit of dust as it was something nobody would ever expect to happen.

The reason why I bring up the support of SQL Server on Linux operating systems now is that preemptive waits that occur inside SQL Server are platform independent. Meaning calls to functions that are only available in the Windows operating system are also recorded when looking at the wait statistics of a SQL-on-Linux instance. The reason why this is possible has everything to do with the underlying technology Microsoft used to bring SQL Server to Linux.

To make SQL Server run on Linux Microsoft adopted a concept called a Platform Abstraction Layer (or PAL for short). The idea of a PAL is to separate the code needed to run, in this case, SQL Server with the code needed to interact with the operating system. Because SQL Server has never run on anything other than Windows, it is full of operating system references inside its code. This would mean that getting SQL Server to run on Linux would end up taking enormous amounts of time because of all the operating system dependencies. So the SQL Server team looked for different approaches to resolve this issue and found its answer in a Microsoft research project called Drawbridge. The definition of Drawbridge can be found on its project page at www.microsoft.com/en-us/research/project/drawbridge/ and reads:

Drawbridge is a research prototype of a new form of virtualization for application sandboxing. Drawbridge combines two core technologies: First, a picoprocess, which is a process-based isolation container with a minimal kernel API surface. Second, a library OS, which is a version of Windows enlightened to run efficiently within a picoprocess

The main part that attracted the SQL Server team to the Drawbridge project was the Library OS technology. This new technology could handle a very wide variety of Windows operating system calls and translate them to the operating system of the host, which in this case is Linux. Now, the SQL Server team did not adapt the Drawbridge technology one-on-one as there were some challenges involved with the research project. One of them was that the research project was officially completed which means there was no support on the project. Another one was a large overlap of technologies inside the SQL Server OS (SOS) and Drawbridge. Both solutions have their own functionalities to handle memory management and threading/scheduling. What eventually was decided was to merge the SQL Server OS and Drawbridge into

a new platform layer called the SQLPAL (SQL Platform Abstraction Layer). Using the SQLPAL the SQL Server team can develop code as they have always done and leave the translation of operating system calls to the SQLPAL. Figure 11-3 shows the interaction between the various layers while running SQL Server on Linux.

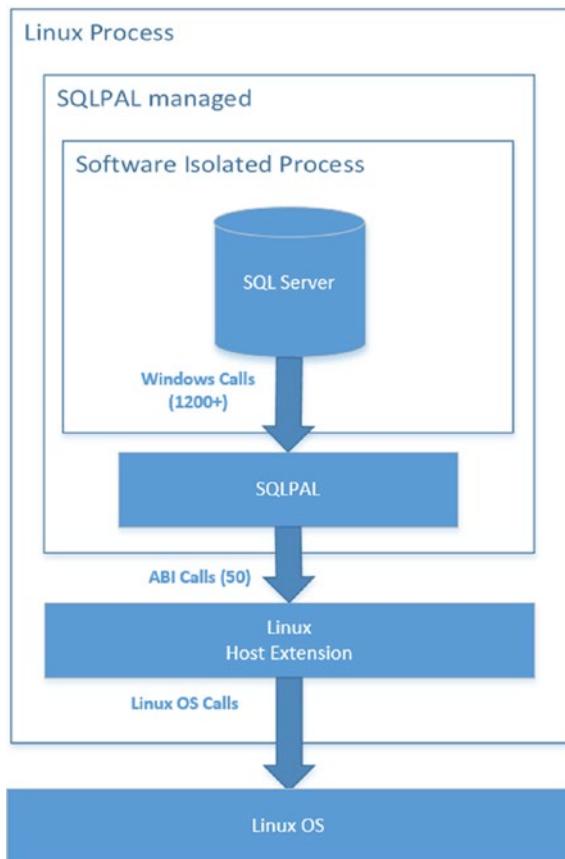


Figure 11-3. PAL layer interaction on SQL-on-Linux

There is a lot more information available on various Microsoft blogs that covers more of the functionality and the design choices of the SQLPAL. If you want to know more about the SQLPAL, or how it came to life, a good recommendation is to read the “SQL Server on Linux: How? Introduction” article over at <https://cloudblogs.microsoft.com/sqlserver/2016/12/16/sql-server-on-linux-how-introduction/>.

For the remainder of this chapter, I will frequently refer to functions used by the Windows operating system. If you are running SQL Server on Linux, remember that the functionality described in this chapter is handled by SQLPAL on Linux but still has the same functionality as on Windows.

PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE

The first preemptive wait types we are going to discuss in this chapter are the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types. As you can probably guess from the wait type names, the functions are related to either encrypting or decrypting messages through the Windows operating system.

What Are the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Wait Types?

As I noted in the previous section, the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types are related to the encryption and decryption of messages. More specifically, they are related to encrypting and decrypting network traffic to and from the SQL Server instance. One case where this is used is when connecting to your SQL Server instance using certificates to encrypt the data that is sent between the client and the SQL Server instance. In that case, SQL Server will need to access the Windows operating system to perform the encryption of the messages that it is sending to the client or to decrypt the messages that are received. The encryption and decryption do not happen inside SQL Server, unlike, for instance, Transparent Data Encryption (TDE), where the encryption/decryption process happens entirely inside SQL Server.

Both the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types do not necessarily indicate any performance problems. They just show you encryption is being used, so there is no real need to troubleshoot these wait types. The overhead of encrypting and decrypting messages is so small that it rarely causes any serious issues (I have yet to come across a case where using certificates to connect to SQL Server caused performance problems).

PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Example

To show you an example of both the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types, I am going to configure a certificate that will be used to encrypt the connection to the SQL Server instance. To make this example reproducible, I have included the steps for creating a self-signed certificate. Normally, in production environments you will use a certificate issued by a certificate authority, but for testing purposes a self-signed certificate is fine.

The first thing I did to be able to generate a self-signed certificate is install Internet Information Services (IIS) on my test virtual machine. IIS makes generating a self-signed certificate very simple.

After the installation of IIS is completed, I open the IIS Manager from Administrative Tools. Then I click the name of my machine and select the Server Certificates option in the Features View, as shown in Figure 11-4.

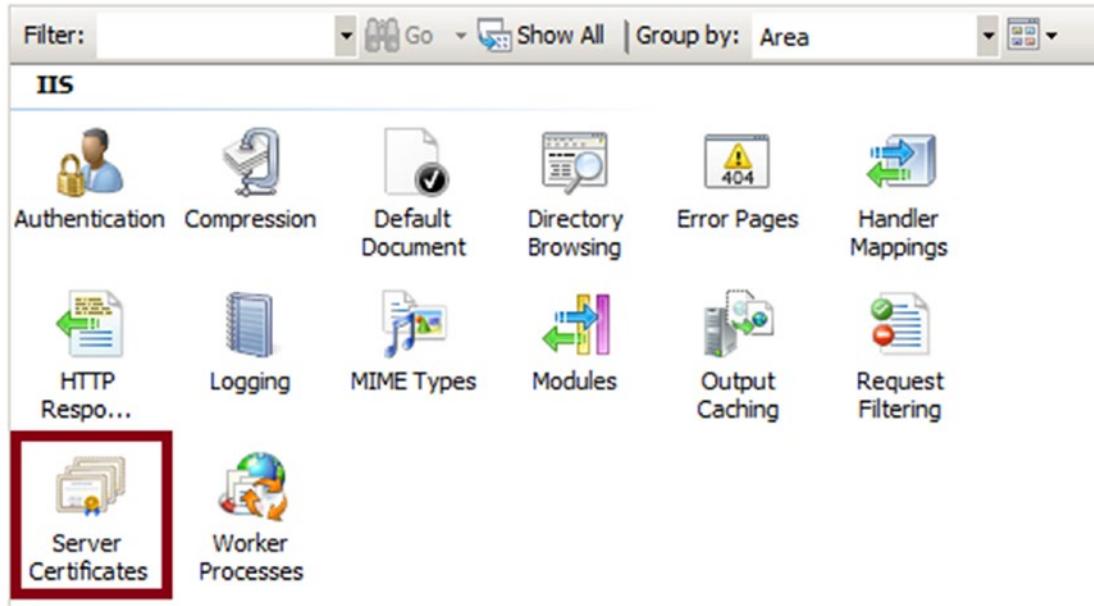


Figure 11-4. Features View inside the IIS Manager

This will open a new Server Certificates view inside the IIS Manager. Inside the Action Pane, I click the Create Self-Signed Certificate option. I am then asked to supply a name for my certificate, so I filled in the name of my test virtual machine as you can see in Figure 11-5.

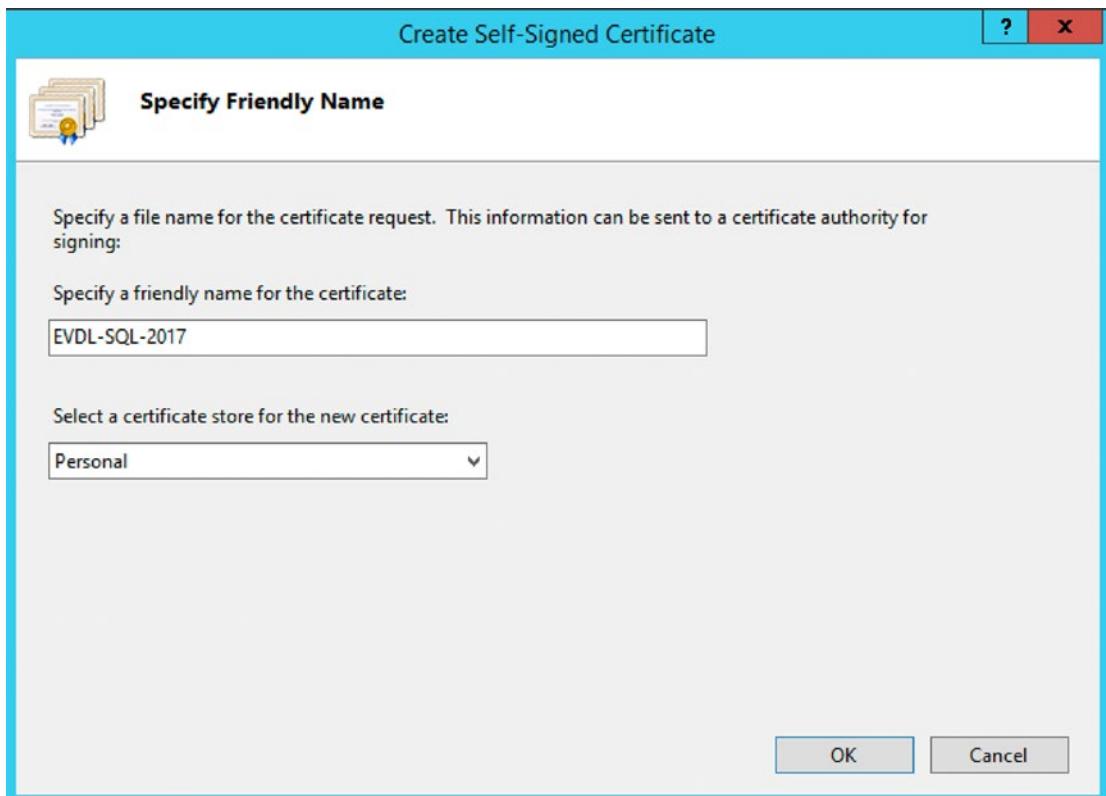


Figure 11-5. Create self-signed certificate

I click OK, and the self-signed certificate will be created and automatically placed inside the correct certificate store on my machine (Local Machine > Personal Certificates).

Now that my self-signed certificate is created and stored inside the certificate store, I need to make sure the account my SQL Server service is running under has permissions to access the certificates. I open MMC by clicking Start > Run, entering MMC, and pressing OK. Now that the MMC console is open, I need to add the Certificates snap-in. I do this by clicking File > Add/Remove Snap-in, selecting the Certificates snap-in, and clicking Add. When prompted for which account I want to manage certificates, I select Computer account, as shown in Figure 11-6, and click Next and Finish.

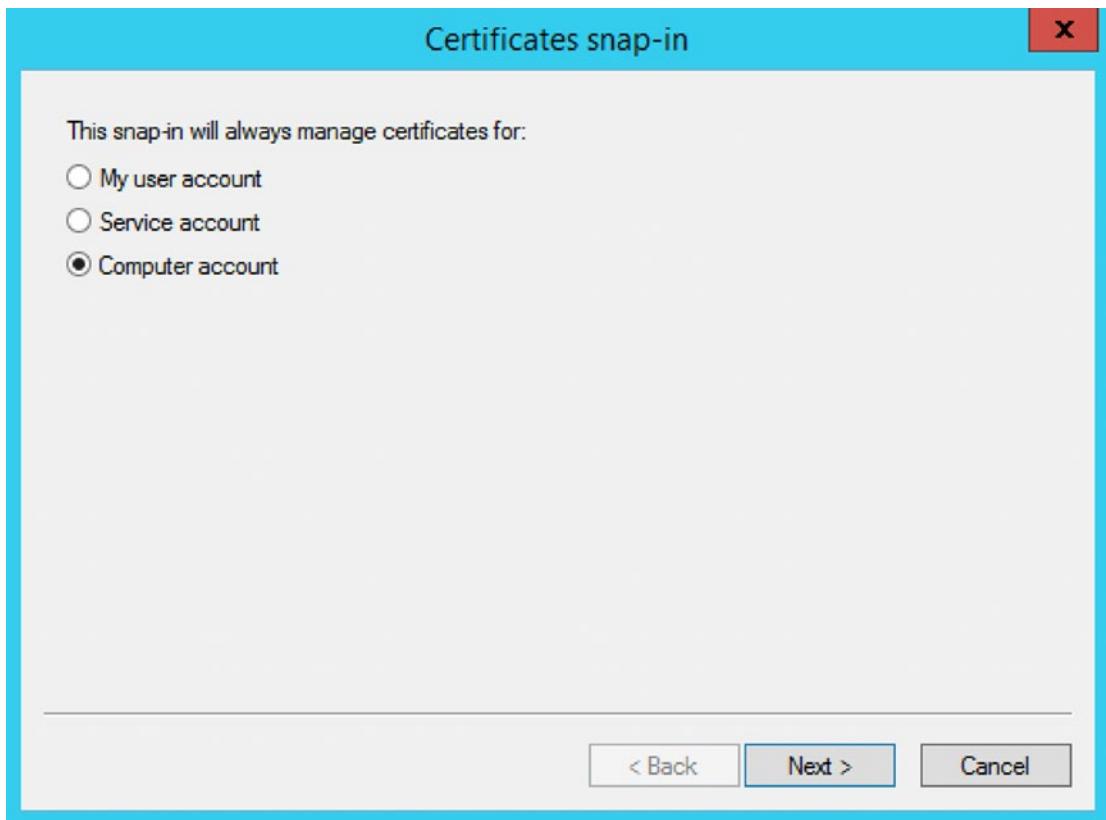


Figure 11-6. Certificate account selection

Inside the Certificates console, I open the folder Certificates (Local Computer) ➤ Personal ➤ Certificates. If the generation of the self-signed certificate inside IIS was correct, I should see the certificate here. Figure 11-7 shows the certificate on my test virtual machine.

Issued To	Issued By	Expiration Date	Intended Purposes	Friendly Name	Status	Certificate Te...
EVDL-SQL2017-01	EVDL-SQL2017-01	10/14/2020	Server Authenticati...	Data Management ...		
EVDL-SQL2017-01	EVDL-SQL2017-01	3/5/2020	Server Authenticati...	EVDL-SQL- 2017		
localhost	localhost	1/25/2024	Server Authenticati...	IIS Express Develop...		

Figure 11-7. Self-signed certificate

I right-click the self-signed certificate and select All Tasks ➤ Manage Private Keys. A permissions dialog opens. Here I need to add the account under which the SQL Server service is running. In my case that is the local administrator user which is added by default. If you run your SQL Server service under a different account, the account only needs read permission on the certificate, as shown in Figure 11-8.

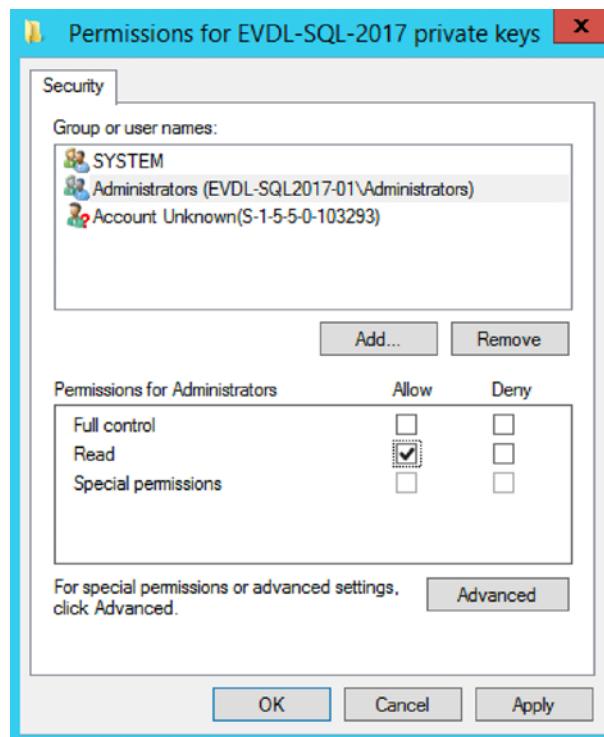


Figure 11-8. Self-signed certificate permissions

After adding the account and selecting the right permission, I click OK to close the dialog. Now that the permissions are correct and the SQL Server Service account can access the certificate, I need to add the self-signed certificate to the network configuration of the SQL Server instance that I want to enable for encryption. I open the SQL Server Configuration Manager and click the SQL Server Network Configuration option. I right-click the SQL Server instance that should use the self-signed certificate and select Properties. Next, I open the Certificates tab and select the self-signed certificate we created earlier. Figure 11-9 shows the dialog on my test virtual machine.

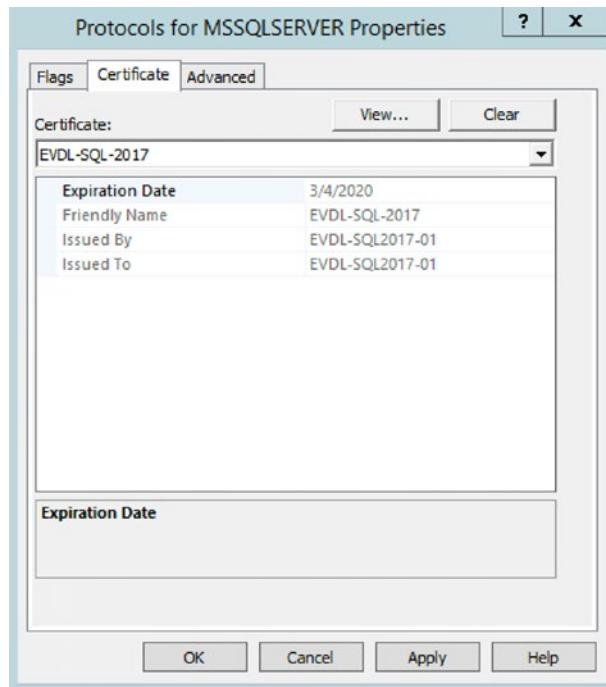


Figure 11-9. Certificate selection

After selecting the self-signed certificate, I click OK to close the dialog. I am notified that the certificate will become active after a restart of the SQL Server service, so I perform a restart of the SQL Server Service.

Right now SQL Server can use the self-signed certificate, but to make sure my network messages are encrypted I have to connect to the SQL Server instance and tell it that I want to use encryption. For this example I will use the SQL Server Management Studio, on the same virtual machine as my SQL Server instance, to connect to the SQL Server instance. If you connect to your SQL Server instance from another machine you need to make sure the self-signed certificate is available on that machine. When the Connect to Server dialog appears inside SQL Server Management Studio, I click the Options button at the bottom right of the dialog. This opens up additional properties for the connection to my SQL Server instance. I select the Encrypt connection checkbox as shown in Figure 11-10, and connect to my SQL Server instance.

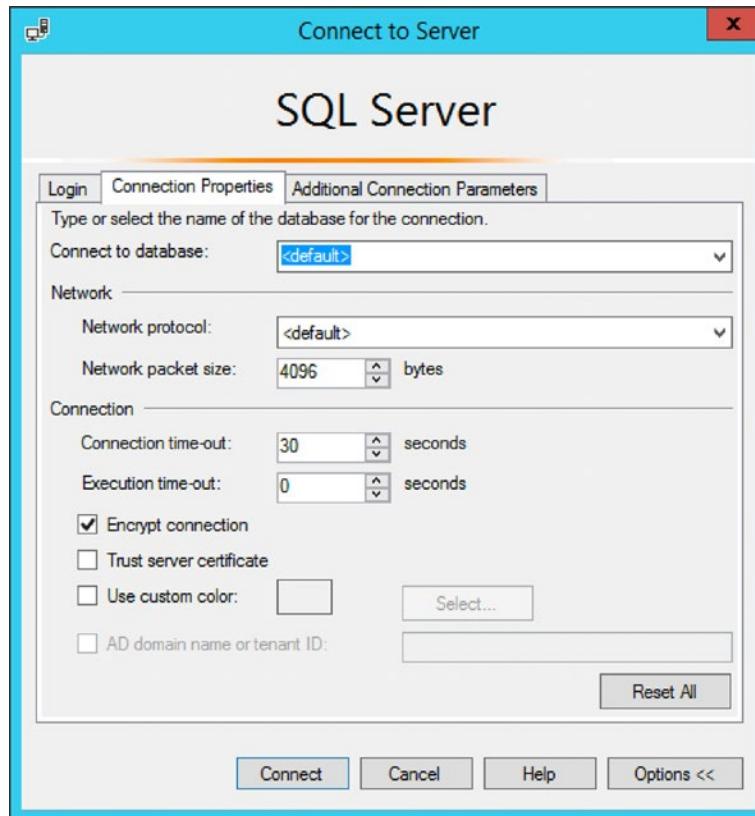


Figure 11-10. Connection Properties in SQL Server Management Studio

Right now I have configured everything I need to make sure SQL Server will use the self-signed certificate to encrypt messages between the SQL Server instance and SQL Server Management Studio, so I can finally take a look at the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types!

Generating the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE waits is very simple now. Basically, every query I execute from the SQL Server Management Studio right now will be encrypted, even if I run SQL Server Management Studio on the same machine as the SQL Server instance. I use the query in Listing 11-1 to reset the sys.dm_os_wait_stats DMV, connect to the AdventureWorks database, perform a simple query, and then look at the waits occurring on the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types.

Listing 11-1. Select query using encrypted connection

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)

USE AdventureWorks
GO

SELECT *
FROM Sales.SalesOrderDetail;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'PREEMPTIVE_OS_ENCRYPTMESSAGE'
OR wait_type = 'PREEMPTIVE_OS_DECRYPTMESSAGE';
```

The results of these queries on my test SQL Server instance are shown in Figure 11-11.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PREEMPTIVE_OS_DECRYPTMESSAGE	1	0	0	0
2	PREEMPTIVE_OS_ENCRYPTMESSAGE	2356	328	2	0

Figure 11-11. PREEMPTIVE_OS_DECRYPTMESSAGE and PREEMPTIVE_OS_ENCRYPTMESSAGE waits

As you can see, the PREEMPTIVE_OS_ENCRYPTMESSAGE wait time has more waits and wait time associated with it. This is logical since I performed a select query and it only had to decrypt the acknowledgment network messages from the client. The results of the query had to be encrypted by SQL Server, which leads to higher waits on the PREEMPTIVE_OS_ENCRYPTMESSAGE wait type.

Lowering PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Waits

Under normal circumstances there should be no need to focus attention on lowering the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types. They mostly just indicate that message encryption is occurring, which is probably a choice that was made when configuring the SQL Server instance. Disabling encryption will dramatically lower the PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait times, but at the cost of security.

PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE Summary

The PREEMPTIVE_OS_ENCRYPTMESSAGE and PREEMPTIVE_OS_DECRYPTMESSAGE wait types indicate that encryption is occurring between the SQL Server instance and a client. These wait types can generally be ignored since they do not directly indicate a performance problem. Lowering them can be achieved by disabling the use of encryption, but this comes at the cost of security.

PREEMPTIVE_OS_WRITEFILEGATHER

The PREEMPTIVE_OS_WRITEFILEGATHER wait type is related to storage interactions, more specifically the writing to files through the Windows operating system.

What Is the PREEMPTIVE_OS_WRITEFILEGATHER Wait Type?

The PREEMPTIVE_OS_WRITEFILEGATHER wait type is related to the WriteFileGather function inside the Windows operating system. If we look up the definition of this function on Books Online, we get the following description: “Retrieves data from an array of buffers and writes the data to a file.” From this description we can assume the function will be called when there is a need to write data to a file. This does not count for every storage subsystem write operation inside SQL Server, however. Generally there is no need for SQL Server to move outside of its own engine to wait for a preemptive operation. There are some exceptions, however, which can result in PREEMPTIVE_OS_WRITEFILEGATHER waits (depending on the Windows function used to perform the storage subsystem interaction). One specific operation inside SQL Server that will always result in PREEMPTIVE_OS_WRITEFILEGATHER waits is the growing of data files. Whenever SQL Server wants to grow a data file, it will need to allocate extra space on the storage subsystem and “zero out” the new space so SQL Server can use it. The allocation of the extra space does not happen inside the SQL Server engine, thus a preemptive operation has to take place, which can lead to preemptive waits on the WriteFileGather function.

PREEMPTIVE_OS_WRITEFILEGATHER Example

To show you an example of PREEMPTIVE_OS_WRITEFILEGATHER waits occurring, I am going to replicate the situation I described in the previous section, growing a database data file. For this example I will restore a backup of the AdventureWorks database; in the SQL Server 2016 version of the database, there exists only a single database data file with a size of 208 MB, as shown in Figure 11-12.

Database files:				
Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize
AdventureW...	ROWS...	PRIMARY	208	By 16 MB, Unlimited
AdventureW...	LOG	Not Applicable	2	By 16 MB, Limited to 209715...

Figure 11-12. Default database file configuration of AdventureWorks (2016 edition)

I am going to grow the single database data file to a size of 10 GB. Because the allocation of the extra space needed for the data file is performed outside SQL Server, this should result in PREEMPTIVE_OS_WRITEFILEGATHER waits.

To perform the action of enlarging the database data file I used the script shown in Listing 11-2. This script will clear the sys.dm_os_wait_stats DMV, enlarge the Adventureworks data file to 800 MB, and then query the sys.dm_os_wait_stats DMV for PREEMPTIVE_OS_WRITEFILEGATHER.

Listing 11-2. Enlarge AdventureWorks database data file

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR)
USE [master]
GO

ALTER DATABASE [AdventureWorks]
MODIFY FILE
(
NAME = N'AdventureWorks2016_Data',
SIZE = 819200KB
);
```

GO

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'PREEMPTIVE_OS_WRITEFILEGATHER';
```

The query in Listing 11-2 is almost instantly completed on my test SQL Server instance, it has very fast storage, and results in the wait information shown in Figure 11-13 for the PREEMPTIVE_OS_WRITEFILEGATHER wait type.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PREEMPTIVE_OS_WRITEFILEGATHER	1	447	447	0

Figure 11-13. PREEMPTIVE_OS_WRITEFILEGATHER waits

Notice that there was only one single wait on the PREEMPTIVE_OS_WRITEFILEGATHER wait type, and the duration was practically as long as it took to perform the enlargement of the data file.

Lowering PREEMPTIVE_OS_WRITEFILEGATHER Waits

When you notice higher-than-normal wait times on the PREEMPTIVE_OS_WRITEFILEGATHER wait type, it means that a process from inside SQL Server is performing actions on the storage subsystem through the Windows operating system. The first matter of action should be to investigate what process initiated the action that resulted in PREEMPTIVE_OS_WRITEFILEGATHER waits. Very frequently this will be the (automatic) growth of a database data or log file. If you allow the data or log files to grow automatically when they are full, you can expect to see PREEMPTIVE_OS_WRITEFILEGATHER waits occur whenever an auto-growth event occurs. This does not necessarily mean there is a problem, but if auto-growth events take a long time to complete because, for instance, the storage subsystem is experiencing performance problems, your queries might experience performance degradation as well.

There is one Windows setting available that I frequently see not configured, instant file initialization. We discussed this setting, and how you can enable it, already in Chapter 6, “IO-Related Wait Types,” under the ASYNC_IO_COMPLETION wait type so I won’t go into detail on how to enable the setting again. Figure 11-14 shows the results

of the query in Listing 11-2 with instant file initialization enabled, and as you can see, the amount of wait time spent on the PREEMPTIVE_OS_WRITEFILEGATHER wait time disappeared completely.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PREEMPTIVE_OS_WRITEFILEGATHER	0	0	0	0

Figure 11-14. PREEMPTIVE_OS_WRITEFILEGATHER waits with instant file initialization turned on

Next to using instant file initialization, the performance of the storage subsystem plays a large part in the PREEMPTIVE_OS_WRITEFILEGATHER wait times. The better your storage subsystem performs, the lower the wait times on the PREEMPTIVE_OS_WRITEFILEGATHER wait type.

Another SQL Server action that can cause higher-than-normal PREEMPTIVE_OS_WRITEFILEGATHER wait times is performing database restores. Much like expending a data file, before SQL Server can restore a database, it needs to allocate free storage for it. This is also related to instant file initialization, which will also speed up database restores just like file enlargements.

PREEMPTIVE_OS_WRITEFILEGATHER Summary

The PREEMPTIVE_OS_WRITEFILEGATHER wait type indicates that SQL Server is asking the Windows operating system to perform an operation on the storage subsystem. Not all operations can be handled from inside the SQL Server engine and actions; for example, the growing of a data file requires the execution of a Windows function to allocate the desired space on the storage subsystem. Instant file initialization is a setting in Windows that can lower the amount of PREEMPTIVE_OS_WRITEFILEGATHER wait time drastically, but the performance of the storage subsystem itself also plays a large role in PREEMPTIVE_OS_WRITEFILEGATHER wait times.

PREEMPTIVE_OS_AUTHENTICATIONOPS

The PREEMPTIVE_OS_AUTHENTICATIONOPS wait type is another preemptive wait type that is related to various Windows authentication functions.

What Is the PREEMPTIVE_OS_AUTHENTICATIONOPS Wait Type?

The PREEMPTIVE_OS_AUTHENTICATIONOPS wait type is recorded whenever SQL Server needs to perform an account authentication, for instance, to authenticate the SQL Server Windows login when it connects to SQL Server. Seeing PREEMPTIVE_OS_AUTHENTICATIONOPS waits occur is to be expected, especially when using mixed-mode authentication and Windows logins inside your SQL Server instance.

One common misconception about the PREEMPTIVE_OS_AUTHENTICATIONOPS wait type is that it is only related to SQL Server logins that use Windows authentication inside a domain. This is not entirely correct. While it is true that PREEMPTIVE_OS_AUTHENTICATIONOPS wait times will frequently be higher when using Active Directory accounts to connect to SQL Server, PREEMPTIVE_OS_AUTHENTICATIONOPS waits will also occur if the SQL Server instance is installed on a machine outside of a domain; the wait times will generally be lower though.

Figure 11-15 shows a simplified image of how SQL Server connects to an Active Directory domain controller to validate the SQL Server Windows login. Keep in mind that the Windows operating system takes care of the communication between the domain controller and the SQL Server, hence the preemptive wait type.

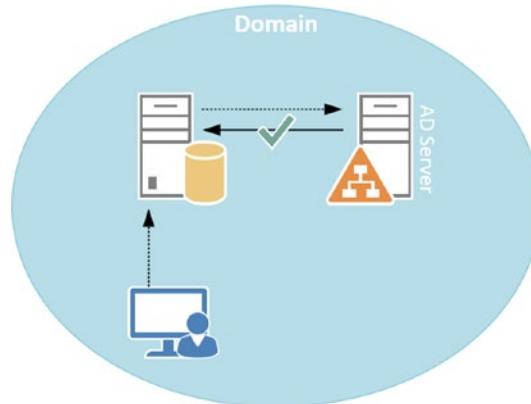


Figure 11-15. SQL Server Windows login authentication inside domain

On a machine that has a SQL Server instance installed but is not part of a domain, the authentication of the Windows login will occur on the machine itself (local accounts).

Because it will generally take a longer time to authenticate a Windows login through a domain controller (the request has to travel across the network and authenticate on another machine), the wait times for the PREEMPTIVE_OS_AUTHENTICATIONOPS wait type will generally be higher for a SQL Server instance inside a domain.

PREEMPTIVE_OS_AUTHENTICATIONOPS Example

To generate an example of PREEMPTIVE_OS_AUTHENTICATIONOPS waits occurring, I do not need to perform any complex actions. Opening a new connection to the SQL Server instance using Windows authentication should be enough. One way to make this easy to measure is by connecting to the SQL Server instance with SQL Server Management Studio, using Windows authentication. Figure 11-16 shows the SQL Server Management Studio connect dialog to my test SQL Server instance. Note that my test SQL Server instance is not inside a domain, and that I use the local administrator account on my machine to connect.

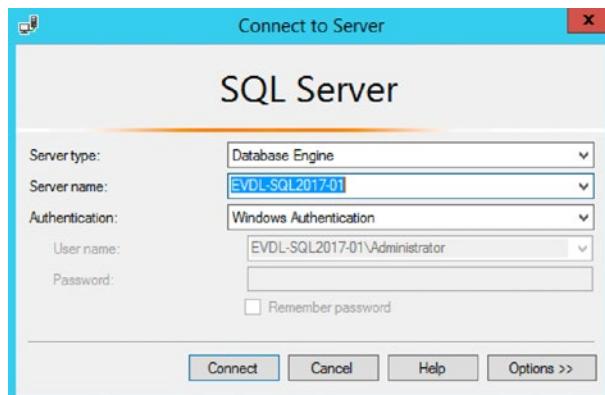


Figure 11-16. Connect SQL Server Management Studio using local Windows authentication

The next step I perform is opening a new Query Window inside SQL Server Management Studio and performing the steps inside the query shown in Listing 11-3.

Listing 11-3. Generate PREEMPTIVE_OS_AUTHENTICATIONOPS waits

```
-- Step 1 Clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- Step 2 Open a new Query Window inside
-- SQL Server Management Studio

-- Step 3 go back to this Query Window
-- and run the query below
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'PREEMPTIVE_OS_AUTHENTICATIONOPS';
```

If you follow the steps commented inside the script in Listing 11-3, you should see PREEMPTIVE_OS_AUTHENTICATIONOPS waits occurring after running the query in step 3. Figure 11-17 shows the results of the query in step 3 on my test machine.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PREEMPTIVE_OS_AUTHENTICATIONOPS	12	1	0	0

Figure 11-17. PREEMPTIVE_OS_AUTHENTICATIONOPS waits

As you can see, the number of waits occurring and their wait times are very low. The point of this example is not to show you an example of very high PREEMPTIVE_OS_AUTHENTICATIONOPS wait times, but rather how they occur naturally when connecting to a SQL Server instance. Because I opened a new Query Window inside SQL Server Management Studio, a new connection to the SQL Server instance will be made using the Windows login I used to connect to my SQL Server instance. Because it is a new connection, the account I used to connect had to be authenticated, resulting in PREEMPTIVE_OS_AUTHENTICATIONOPS waits.

Lowering PREEMPTIVE_OS_AUTHENTICATIONOPS Waits

In the example, I have shown you how PREEMPTIVE_OS_AUTHENTICATIONOPS waits occur naturally whenever you connect to a SQL Server instance. Now imagine a situation where your SQL Server instance is part of a domain environment and it uses Windows authentication to authenticate domain users (or groups) against an Active Directory. In that case your authentication request has to travel across the network in order to perform

the authentication of the account. There are many factors involved that can impact the speed of the authentication request; for instance, if your Domain Controller is under a lot of stress, it can take longer to perform the authentication, or if your network experiences performance degradation, it will also impact the authentication request. These factors also contribute to the PREEMPTIVE_OS_AUTHENTICATIONOPS wait type and can result in higher wait times.

I would like to describe to you a case I encountered at a client that involved the PREEMPTIVE_OS_AUTHENTICATIONOPS wait type to give you an idea of how you can lower PREEMPTIVE_OS_AUTHENTICATIONOPS wait times.

At this client they used an application that connected to SQL Server using the Windows account that was logged in on the computer that ran the application. The computers and the SQL Server were all part of a domain. From a security perspective, the application was well designed, as it did not require separate SQL Server users who needed permission on the database and also didn't use a generic account to connect to the SQL Server instance and execute queries. Inside, the database-specific objects (like tables) were also secured based on domain users and groups.

The client started to experience server performance problems inside the application after deploying it to every (3000+) computer inside the company. The DBA at the client couldn't find any problems, there were no infrastructure-related performance problems on the SQL Server instance, and executing the queries on the SQL Server instance itself revealed no issues. When we looked at the wait statistics, we noticed that the most prevalent wait type was the PREEMPTIVE_OS_AUTHENTICATIONOPS wait type. We also noticed that the application would connect to the SQL Server instance, run a query, then disconnect again. Because so many concurrent users were using the application, it resulted in a high amount of Windows authentication requests, so many that the Domain Controller couldn't handle them, resulting in the slower processing of authentication requests.

In this case the Domain Controller was a virtual machine, and after adding more processor and memory resources, it was able to keep up with the high amount of authentication requests.

As you can see from this case, seeing high PREEMPTIVE_OS_AUTHENTICATIONOPS wait times does not necessarily mean your SQL Server instance is running into problems, especially in a domain environment, as the performance of your Domain Controller also plays a large role in PREEMPTIVE_OS_AUTHENTICATIONOPS wait times.

The moral of the story is, if you notice higher-than-normal PREEMPTIVE_OS_AUTHENTICATIONOPS wait times, you will need to investigate much more than the SQL Server instance. Make sure to check the performance of your Domain Controllers if you are using Windows authentication inside a domain. Check every infrastructure part between your SQL Server instance and the Domain Controller, like network switches, firewalls, and so on. All of these infrastructure parts will add additional latency for each authentication request, which will result in higher PREEMPTIVE_OS_AUTHENTICATIONOPS wait times, making it a difficult wait type to troubleshoot.

PREEMPTIVE_OS_AUTHENTICATIONOPS Summary

The PREEMPTIVE_OS_AUTHENTICATIONOPS wait type is related to performing authentication requests by the Windows operating system. It is normal to see PREEMPTIVE_OS_AUTHENTICATIONOPS waits occur, especially when your SQL Server instance is part of a domain and uses Windows authentication to authenticate users. Higher-than-normal wait times can indicate that authentication requests are taking longer than normal to complete. This does not necessarily mean that your SQL Server instance is running into a performance problem. If the Domain Controller cannot process the authentication requests fast enough, it will result in higher PREEMPTIVE_OS_AUTHENTICATIONOPS wait times. A slow network connection to the Domain Controller, firewall, or switch configurations can also impact PREEMPTIVE_OS_AUTHENTICATIONOPS wait times.

PREEMPTIVE_OS_GETPROCADDRESS

The final wait type I would like to discuss in this chapter is the PREEMPTIVE_OS_GETPROCADDRESS wait type. The PREEMPTIVE_OS_GETPROCADDRESS wait type is related to the execution of extended stored procedures inside SQL Server.

Extended stored procedures allow you to create external routines in a language other than T-SQL; for instance, using the C# programming language. These extended stored procedures are loaded into SQL Server using .dll files and can expand the capabilities of SQL Server programming by allowing you to perform actions that would be impossible in T-SQL, like reading/writing Windows Registry entries.

Extended stored procedures are marked as deprecated since SQL Server 2008 and Common Language Runtime (CLR) should be used instead of them. However, there are still cases inside SQL Server that require extended stored procedures, and some third-party software vendors still rely on them.

What Is the PREEMPTIVE_OS_GETPROCADDRESS Wait Type?

The PREEMPTIVE_OS_GETPROCADDRESS wait type is recorded whenever the entrypoint inside an extended stored procedure is loaded. The entrypoint is called upon whenever SQL Server loads or unloads the extended stored procedure .dll file. Under normal conditions the loading of the entrypoint should complete quickly, resulting in very low PREEMPTIVE_OS_GETPROCADDRESS wait times, if any, but depending on the extended stored procedure, or problems related to loading the extended stored procedure .dll file, it is possible to notice higher wait times. Important to keep in mind is that the PREEMPTIVE_OS_GETPROCADDRESS wait type only records the time it took to load the entrypoint of the .dll file, not the execution time of the extended stored procedure. Figure 11-18 shows a (simplified) overview of how extended stored procedures are executed by SQL Server.

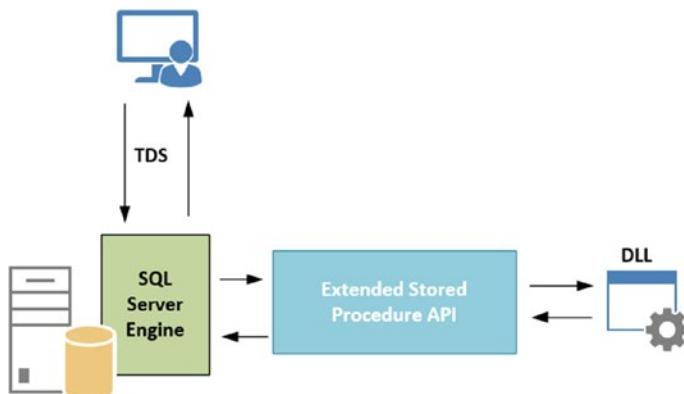


Figure 11-18. Executing an extended stored procedure

Not only can you write your own extended stored procedures to perform actions not possible using T-SQL, SQL Server itself comes shipped with many different extended stored procedures. Most of these can be recognized by the `xp_` prefix in the extended stored procedure name, though not all of them have this prefix. Figure 11-19 shows a selection of extended stored procedures inside the `master` database of my test SQL Server instance.

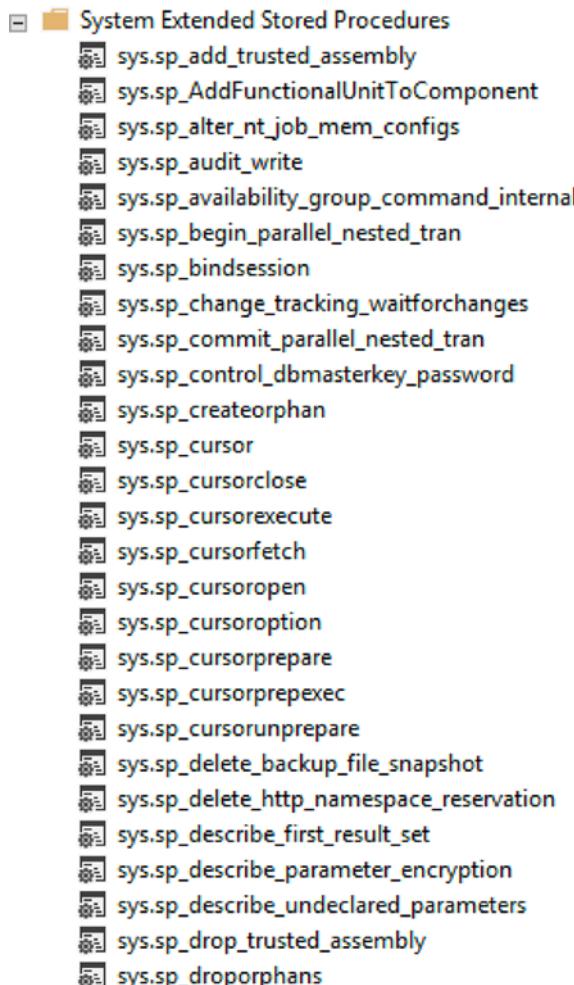


Figure 11-19. Selection of extended stored procedures inside the master database

Probably the most notorious extended stored procedure is the `xp_cmdshell` extended stored procedure. The `xp_cmdshell` extended stored procedures makes it possible to execute a command inside a Windows command shell from within SQL Server. This is a huge security risk if your SQL Server instance is compromised, since it gives access to commands that can affect the entire Windows operating system. Thankfully, it is impossible to run the `xp_cmdshell` extended stored procedure by default; you have to specifically allow its use by configuring an advanced configuration setting.

PREEMPTIVE_OS_GETPROCADDRESS Example

For this example I will execute an extended stored procedure already present inside SQL Server, `xp_getnetname`, instead of writing a custom extended stored procedure, which is far beyond the scope of this book. The `xp_getnetname` is an undocumented extended stored procedure that returns the NETBIOS name of the machine that hosts your SQL Server instance. Before executing the `xp_getnetname` extended stored procedure, I clear the `sys.dm_os_wait_stats` DMV, and after the execution of `xp_getnetname`, I query the DMV for `PREEMPTIVE_OS_GETPROCADDRESS` wait information. Listing 11-4 shows the entire query I executed on my test SQL Server instance.

Listing 11-4. Execute `xp_getnetname` and query wait statistics

```
USE [master]
GO

DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

exec xp_getnetname;

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'PREEMPTIVE_OS_GETPROCADDRESS';
```

The results of the query in Listing 11-4 can be seen in Figure 11-20.

	Server Net Name			
1	EVDL-SQL2017-01			
wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PREEMPTIVE_OS_GETPROCADDRESS	1	0	0

Figure 11-20. `PREEMPTIVE_OS_GETPROCADDRESS` wait

The results aren't spectacular. Apparently, the `xp_getnetname` extended stored procedure doesn't cause any problems when loading the .dll entrypoint, since there is no wait time recorded. A wait still did occur though, as you can see in the `waiting_tasks_count` column, it just took SQL Server less than a millisecond to load the entrypoint.

Lowering PREEMPTIVE_OS_GETPROCADDRESS Waits

Since the PREEMPTIVE_OS_GETPROCADDRESS wait type is directly related to executing extended stored procedures, the first step in your investigation should be to detect what extended stored procedure is being executed and what its function is.

I have seen PREEMPTIVE_OS_GETPROCADDRESS waits occur at a number of clients because they were using a third-party backup application that used extended stored procedures to perform a database backup, but there are many more possible causes for high PREEMPTIVE_OS_GETPROCADDRESS wait times. Knowing which extended stored procedure is being executed can help you trace what process is executing the extended stored procedure.

There have also been some known bugs inside SQL Server 2008 and 2008R2 that reported higher-than-normal PREEMPTIVE_OS_GETPROCADDRESS wait times because the execution time of the extended stored procedure was also recorded in the wait times, instead of only the entrypoint loading. If you are still using SQL Server 2008 or 2008R2 and experience very high PREEMPTIVE_OS_GETPROCADDRESS wait times, it might be worth your while to upgrade to the latest Service Pack and check if the PREEMPTIVE_OS_GETPROCADDRESS wait times go down. Or even better, upgrade to a higher version of SQL Server since SQL Server 2008R2 is marked end-of-life as of July 9, 2019.

PREEMPTIVE_OS_GETPROCADDRESS Summary

The PREEMPTIVE_OS_GETPROCADDRESS wait type is directly related to the execution of extended stored procedures. Extended stored procedures can be written in a variety of programming languages like C# and allow you to perform actions that would otherwise be impossible in T-SQL. Wait time for the PREEMPTIVE_OS_GETPROCADDRESS wait type is recorded whenever the entrypoint inside an extended stored procedure .dll is loaded. In normal situations wait times for the PREEMPTIVE_OS_GETPROCADDRESS wait type are very low. Seeing high PREEMPTIVE_OS_GETPROCADDRESS wait times can indicate that the entrypoint loading is running into problems. There have also been bugs related to the calculation of the PREEMPTIVE_OS_GETPROCADDRESS wait type inside SQL Server 2008 and 2008R2. If you are running SQL Server 2008 or 2008R2 and experience high PREEMPTIVE_OS_GETPROCADDRESS wait times, it might be worth your time to upgrade to the latest Service Pack or move to a higher version of SQL Server since SQL Server 2008R2 is marked end-of-life as of July 9, 2019.

CHAPTER 12

Background and Miscellaneous Wait Types

SQL Server has many different internal processes that can run into a wait of a specific wait type, and so far we have discussed quite a few of them. Some of these internal processes are constantly running inside SQL Server, waiting until there is work for them to do. While these processes, frequently called *background processes*, are waiting for work to arrive, SQL Server will record the time they are waiting for work as wait time on specific wait types related to these background processes. While these background wait types are not directly related to performance problems, they frequently have the highest wait time and will show up at the top of the wait time list when you query the top wait types ordered by wait time.

Frequently these background wait types are called *benign*, and can safely be ignored because they simply indicate that an internal process is waiting for work to arrive. This logic is also true for the wait types we will discuss in this chapter, but instead of just telling you to ignore them when analyzing wait statistics, I want to give you some background information about them so you know what they measure and why it is safe to ignore them. Keep in mind that we are still talking about SQL Server here, which means that “it depends” on many factors as to whether you can completely ignore these background wait types. You wouldn’t be the first person to run into a performance problem and only to find out that an ignored background process was actually the cause of the issue. So my advice is to ignore but to not forget about them!

Next to the background wait types I also added a number of miscellaneous wait types that were difficult to place in an earlier chapter because they didn’t quite fit in with the chapter’s wait type category.

Since the background wait types inside this chapter record wait time constantly when their associated processes are waiting for work to do, I did not include an example section or a lowering wait time section for these wait types.

CHECKPOINT_QUEUE

The first wait type in this chapter is one of those background wait types that accumulates large amounts of wait time over time CHECKPOINT_QUEUE. The CHECKPOINT_QUEUE wait type can in many cases be safely ignored, but understanding what the wait type stands for and why it has such high wait times can't hurt.

What Is the CHECKPOINT_QUEUE Wait Type?

The CHECKPOINT_QUEUE wait type is related to the checkpoint process in SQL Server that is responsible for writing “dirty” (modified) data pages from the buffer cache to the data file on disk. In Chapter 6, “IO-Related Wait Types,” we took a good look at the checkpoint process when we discussed the SLEEP_BPOOL_FLUSH wait type, so I won’t repeat all the information again here. What is important to know, and the reason why this wait type can normally be ignored, is that the CHECKPOINT_QUEUE wait type indicates that the checkpoint process is waiting for work. This means that wait times on the CHECKPOINT_QUEUE wait type don’t indicate any performance issues; they just indicate the time the checkpoint processes spent waiting on work. On SQL Server instances that aren’t very busy, or don’t see many data modification operations, the wait time can reach very high values.

The recording of CHECKPOINT_QUEUE wait times inside the sys.dm_os_wait_stats and sys.dm_os_waiting_tasks DMVs goes through a specific internal routine that might return unexpected wait times (like sudden spikes inside your baseline). Figure 12-1 shows the results of queries against the sys.dm_os_wait_stats and sys.dm_os_waiting_tasks DMVs for wait information of the CHECKPOINT_QUEUE wait type on my test SQL Server instance.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms	
1	CHECKPOINT_QUEUE	0	0	0	0	
	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x00000098FC01DC28	14	0	2110373	CHECKPOINT_QUEUE	0x00000099C611EB00

Figure 12-1. CHECKPOINT_QUEUE waits

What is interesting to notice here is that the cumulative wait times inside the sys.dm_os_wait_stats DMV stay at 0, while the wait times inside the sys.dm_os_waiting_tasks DMV have a very high value. My test SQL Server instance doesn’t perform much work in

the background, so it is logical that the checkpoint process spends most of its time waiting for work. The reason for the difference in wait times between both DMVs is related to the way SQL Server executes checkpoint operations. The wait times shown in both the DMVs are only recorded by the automatic checkpoint process. A manual checkpoint execution does not impact the wait times. As part of the automatic checkpoint process, the wait times of the sys.dm_os_waiting_tasks DMV are moved to the sys.dm_os_wait_stats DMV and reset to 0. So, if you notice very high CHECKPOINT_QUEUE wait times inside the sys.dm_os_waiting_tasks, it means it was some time ago that the automatic checkpoint process ran.

To show you a simple demonstration of this behavior, I created a table, reset the sys.dm_os_wait_stats DMV, inserted a few rows inside the table, performed a manual checkpoint, and queried the sys.dm_os_wait_stats and sys.dm_os_waiting_tasks DMV, as shown in Listing 12-1.

Listing 12-1. CHECKPOINT_QUEUE example

```
-- Create a table in the AdventureWorks database
USE [AdventureWorks]
GO

CREATE TABLE check_test
(
    ID UNIQUEIDENTIFIER,
    RandomData VARCHAR(50)
);
GO

-- Clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- Insert a few rows into our table
INSERT INTO check_test
(
    ID,
    RandomData
)
VALUES
```

CHAPTER 12 BACKGROUND AND MISCELLANEOUS WAIT TYPES

```
(  
NEWID(),  
CONVERT(varchar(50), NEWID())  
);  
GO 100  
  
CHECKPOINT 1;  
  
-- Query Wait Statistics  
SELECT *  
FROM sys.dm_os_wait_stats  
WHERE wait_type = 'CHECKPOINT_QUEUE';  
  
SELECT *  
FROM sys.dm_os_waiting_tasks  
WHERE wait_type = 'CHECKPOINT_QUEUE';
```

Figure 12-2 shows the results of the queries made against the sys.dm_os_wait_stats and sys.dm_os_waiting_tasks DMVs.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	CHECKPOINT_QUEUE	0	0	0	0

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x00000098FC01DC28	14	0	2206318	CHECKPOINT_QUEUE	0x00000099C611EB00

Figure 12-2. CHECKPOINT_QUEUE waits

As you can see, the manual checkpoint didn't generate any waits inside the sys.dm_os_wait_stats DMV. Also, an automatic checkpoint didn't occur, because inserting 100 rows generated too few log records to trigger an automatic checkpoint.

If we were to insert more rows, we should be able to trigger an automatic checkpoint. In this case I ran the following query to insert 100,000 rows into the table we created in Listing 12-1. While the insert was running, I queried the sys.dm_os_wait_stats and sys.dm_os_waiting_tasks DMVs repeatedly to see if anything changed. See the following:

```

INSERT INTO check_test
(
ID,
RandomData
)
VALUES
(
NEWID(),
CONVERT(varchar(50), NEWID())
);
GO 100000

```

After a few seconds I noticed that the wait time for the CHECKPOINT_QUEUE wait type was moved to the sys.dm_os_wait_stats DMV, as shown in Figure 12-3.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms	
1	CHECKPOINT_QUEUE	1	2267469	2267469	0	
	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x00000098FC01DC28	14	0	931	CHECKPOINT_QUEUE	0x00000099C611EB00

Figure 12-3. CHECKPOINT_QUEUE waits

Apparently, we inserted enough log records to cause an automatic checkpoint to occur, and as you can see from this example, only an automatic checkpoint will write the wait times of the CHECKPOINT_QUEUE to the sys.dm_os_wait_stats DMV.

You should keep this behavior in mind when you notice sudden, very high wait time values inside the sys.dm_os_wait_stats DMV. This will normally only occur in SQL Server instances that either have a very small workload or that have a workload mainly consisting of read operations instead of data modification operations.

CHECKPOINT_QUEUE Summary

The CHECKPOINT_QUEUE wait type is related to checkpoint operations inside SQL Server. Wait time on the CHECKPOINT_QUEUE wait type is recorded while SQL Server is waiting for an automatic checkpoint operation to take place. This is one of the wait types you can normally safely ignore because it doesn't indicate there are any performance issues. The wait times on the CHECKPOINT_QUEUE wait type are recorded differently between the

`sys.dm_os_wait_stats` and `sys.dm_os_waiting_tasks` DMV, and this can cause sudden high wait times when querying the `sys.dm_os_wait_stats` DMV. Keep this behavior in mind when noticing high `CHECKPOINT_QUEUE` wait times inside the `sys.dm_os_wait_stats` DMV.

DIRTY_PAGE_POLL

The `DIRTY_PAGE_POLL` wait type was introduced in SQL Server 2012 with the indirect checkpoint feature and behaves a lot like the previous wait type we discussed, `CHECKPOINT_QUEUE`. While the automatic checkpoint process runs at a set interval of 1 minute, the indirect checkpoint feature allows you to configure a specific checkpoint interval on a per-database basis. Even if you are not using indirect checkpoint, the `DIRTY_PAGE_POLL` wait type will still accumulate wait time.

What Is the DIRTY_PAGE_POLL Wait Type?

The `DIRTY_PAGE_POLL` wait type is another background wait that can normally be safely ignored. The wait type is related to the recovery writer process that is used by the indirect checkpoint feature that runs continuously in the background of your SQL Server instance. Because of this connection, let's take a quick look what indirect checkpoints are and how they work.

As we know, the checkpoint process inside SQL Server is responsible for writing modified data pages from the buffer cache to the database data file on disk. By default, the checkpoint process runs automatically every minute, or when enough log records have been generated. The checkpoint process plays a vital part in the recovery duration of your SQL Server databases when a crash occurs. Take, for instance, the following scenario: while you are performing many modifications to a database in your SQL Server instance, a crash occurs. Luckily, you were able to simply restart the SQL Server service to get everything up and running again. The first thing SQL Server will do is start a recovery process. The recovery process will check the transaction log for any transactions that were not committed when the crash occurred and perform a rollback of the transaction. The recovery process will also check whether any data pages that were modified by a committed transaction received their modification inside the database data file because they were only modified in the buffer cache.

If any of those pages are found, SQL Server will use the transaction log to redo these transactions. Now imagine you have a busy SQL Server instance where many thousands of modifications are performed every minute. This means that the chance that there is a high number of dirty pages not written to disk yet is pretty high. If your SQL Server then crashes (or, for instance, a failover occurs), the recovery process will take more time to complete. Indirect checkpoints can help us keep this recovery process as short as possible. By configuring this feature we can tell SQL Server to write modified data pages to disk faster; for instance, every 10 seconds. Figure 12-4 shows the location and name of the indirect checkpoint feature inside the properties of a database.

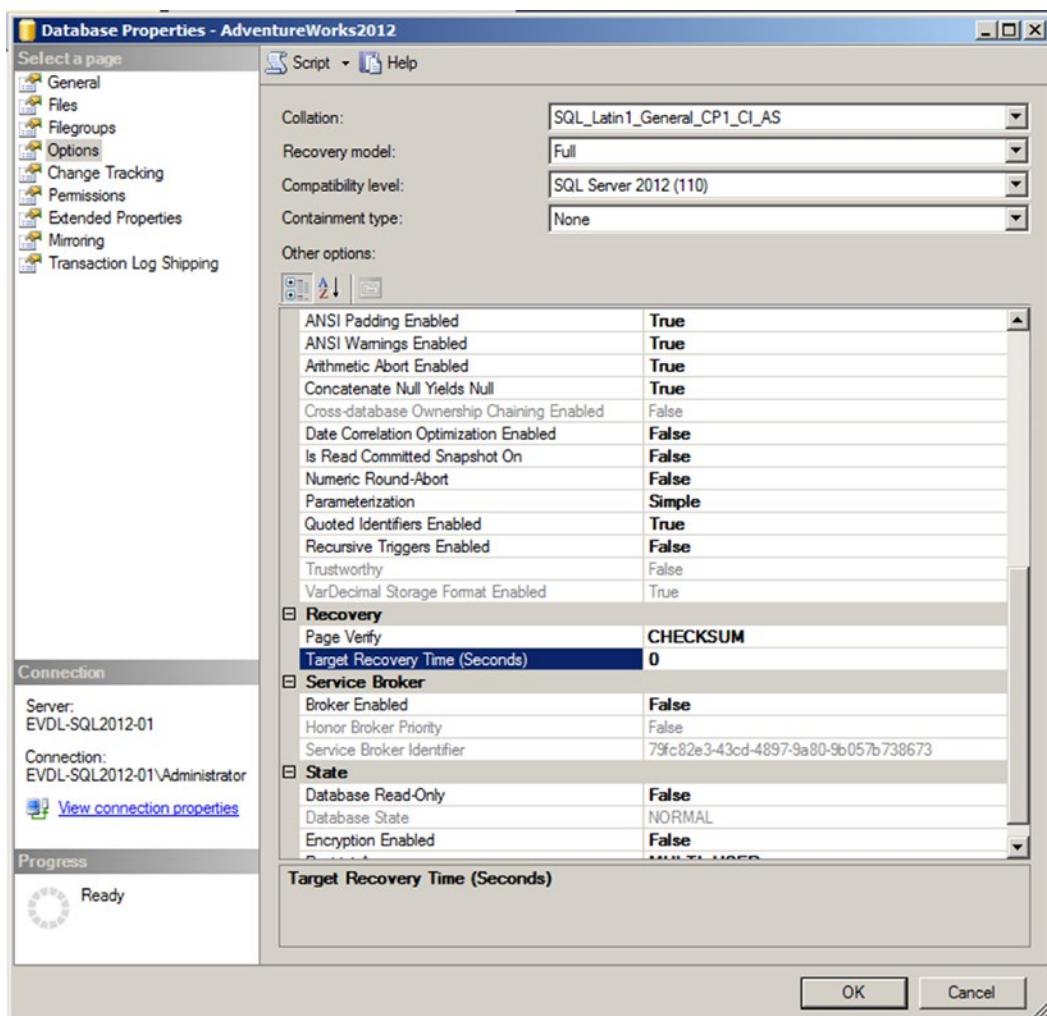


Figure 12-4. Indirect checkpoint feature location and value

By default, the value of the **Target Recovery Time (Seconds)** configuration option is 0. This means that indirect checkpoints are not being used. If you modify the value to anything other than 0, an indirect checkpoint will occur at the interval in seconds you specified.

Starting from SQL Server 2016, indirect checkpoints are automatically configured whenever you create a new database inside the SQL Server instance. In those cases the **Target Recovery Time (Seconds)** will be set to a value of 60 instead of 0.

The time you configure in the **Target Recovery Time (Seconds)** option does not mean that every x seconds the checkpoint process will be executed, however. By setting this value SQL Server will calculate how many dirty pages can exist before they need to be written to the database data file so that the recovery process never takes longer than the time specified. So, for instance, if you configure the **Target Recovery Time (Seconds)** option to 15 seconds, SQL Server will write dirty pages to the database data file at such an interval that when the SQL Server instance fails it can be recovered within 15 seconds.

To monitor how many dirty pages are inside the buffer cache so SQL Server knows when the dirty-page threshold has been reached, the recovery writer was introduced. Even if you do not configure the **Target Recovery Time (Seconds)** option, **DIRTY_PAGE_POLL** waits will still occur because the recovery writer process will still poll the number of dirty pages inside the buffer cache, even though no action is taken upon that number. As you can see in Figure 12-5, the wait times can reach high values easily even when not using indirect checkpoints.

```
-- Query Wait Statistics
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'DIRTY_PAGE_POLL';
```

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	DIRTY_PAGE_POLL	5458	596818	120	35

Figure 12-5. DIRTY_PAGE_POLL waits

Indirect checkpoints also have a risk associated with them. Configuring the **Target Recovery Time (Seconds)** option to a very low value can lead to extra load on the storage subsystem because dirty pages are continuously written to disk. Be sure to test the setting extensively before configuring it on your production SQL Server instances.

DIRTY_PAGE_POLL Summary

The DIRTY_PAGE_POLL wait type was introduced in SQL Server 2012 with the introduction of the indirect checkpoints feature (which ended up being the default setting for new databases created in SQL Server 2016 or higher). Even if you do not use indirect checkpoints, the DIRTY_PAGE_POLL wait type will still accumulate wait time because of the new recovery writer process. Normally the DIRTY_PAGE_POLL wait type does not indicate a performance problem, and as such it can safely be ignored when analyzing wait statistics on your SQL Server instance.

LAZYWRITER_SLEEP

The LAZYWRITER_SLEEP wait type is, surprise, related to the SQL Server internal lazywriter process. The lazywriter process shares some similarities with the checkpoint process we discussed earlier in this chapter, in that it also writes dirty pages from the buffer cache to the database data file. The similarities end here, though, because the reason why the lazywriter process writes these pages to the database data file is completely different than the checkpoint process.

What Is the LAZYWRITER_SLEEP Wait Type?

Just like with other wait types we have discussed so far in this chapter, the LAZYWRITER_SLEEP wait type occurs when an internal SQL Server process, in this case the lazywriter process, is waiting for work. The lazywriter process is a background process that will become active at a certain time interval. When it becomes active it will scan the size of the buffer cache and determine if there are enough free pages inside the buffer cache. It is important that there are always a certain number of free pages inside the buffer cache so that new page requests can fit directly without first having to swap out other pages. If the lazywriter process determines there are enough free pages in the buffer cache,

it will go back to sleep again and record the LAZYWRITER_SLEEP wait type while it is sleeping. However, if there are not enough free pages inside the buffer cache, the lazywriter process will detect, between checkpoints, which dirty pages in the buffer cache haven't been accessed for a while, write them to the database data file, and remove them from the buffer cache. So, if there are more than enough free pages inside the buffer cache, the lazywriter process doesn't have much work to do. If your SQL Server instance is under memory pressure, the lazywriter process will be far busier while swapping out dirty pages and freeing up room inside the buffer cache. Figure 12-6 shows the relationship of the checkpoint and lazywriter processes with a flowchart.

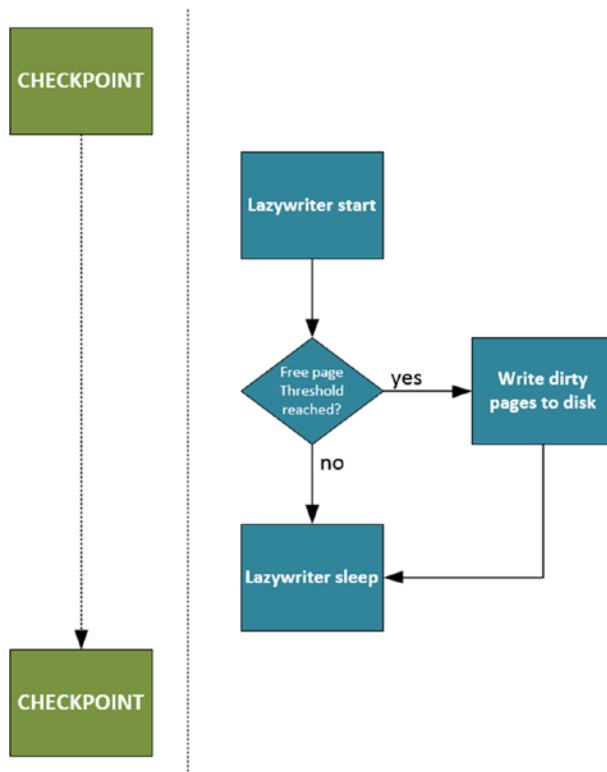


Figure 12-6. Checkpoint and lazywriter processes

Because the LAZYWRITER_SLEEP wait type indicates the time the lazywriter process spends sleeping, or waiting for work, it is another one of those wait types you can safely ignore. There is a catch however—if the lazywriter process is constantly working to move dirty pages from the buffer cache to the database data file, it can indicate your SQL

Server instance is experiencing memory pressure. This is bad for performance because every page has to be moved to the buffer cache before it can get read or modified. This behavior can potentially result in lower-than-normal wait times on the LAZYWRITER_SLEEP wait type.

LAZYWRITER_SLEEP Summary

The LAZYWRITER_SLEEP wait type is related to the lazywriter internal SQL Server process. The lazywriter process starts at a fixed time interval and is responsible for writing dirty data pages to the database data file if there are not enough free pages available inside the buffer cache. The LAZYWRITER_SLEEP wait type indicates that the lazywriter process is currently not running, or is sleeping, until it is signaled to wake up and check the buffer cache. Because the LAZYWRITER_SLEEP wait type only shows us how much time the lazywriter process spends being inactive, it can in most cases be ignored.

MSQL_XP

In the last section of Chapter 11, “Preemptive Wait Types,” we discussed the PREEMPTIVE_GETPROCADDRESS wait type. We learned that the PREEMPTIVE_GETPROCADDRESS wait type records wait time when the entrypoint of an extended stored procedure is loaded. One important thing I noted was that the PREEMPTIVE_GETPROCADDRESS wait type does not record the execution time of the extended stored procedure, only the entrypoint loading. The execution time of an extended stored procedure is actually tracked by another wait type, MSQL_XP.

What Is the MSQL_XP Wait Type?

The MSQL_XP wait type records the execution time of extended stored procedures on your SQL Server instance. The MSQL_XP wait type is also used to detect deadlock situations when using Multiple Active Result Sets (MARS). MARS is a feature that allows the execution of multiple (concurrent) batches through a single SQL Server connection. We won’t go further into detail about MARS, but you can find some more information about it here: <https://msdn.microsoft.com/en-us/library/ms131686.aspx>.

The most common reason for seeing higher-than-normal wait times on the MSQL_XP wait type is the execution of extended stored procedures. This does not necessarily mean there is a problem as long as the execution time of the extended stored procedures stays the same. However, if an extended stored procedure takes more time than expected, you are sure to notice it in the increase of the MSQL_XP wait time when comparing the wait time against a baseline.

MSQL_XP Example

To demonstrate that MSQL_XP waits occur when extended stored procedures are being executed, I created a simple example using the script in Listing 12-2. The script will reset the sys.dm_os_wait_stats DMV, execute an extended stored procedure (in this case the xp_dirtree extended stored procedure inside the master database), and query the sys.dm_os_wait_stats for MSQL_XP wait information.

Listing 12-2. MSQL_XP example

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
EXEC master..xp_dirtree 'c:\windows';
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'MSQL_XP';
```

The results of the query in Listing 12-2 on my test SQL Server instance can be seen in Figure 12-7. The top window shows the xp_dirtree results, the bottom window the results of the query against the sys.dm_os_waits_stats DMV.

	subdirectory	depth
1	ADFS	1
2	ar	2
3	bg	2
4	cs	2
5	da	2
6	de	2
7	el	2
8	en	2

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	MSQL_XP	1	21224	21224	0

Figure 12-7. MSQL_XP wait

The information in the `sys.dm_os_wait_stats` DMV shows us that one wait occurred on the `MSQL_XP` wait type with a wait time of 21,224 milliseconds. This is almost identical to the time it took to execute the query in Listing 12-2, which was 21 seconds on my test SQL Server instance.

Lowering MSQL_XP Waits

When noticing higher-than-normal wait times for the `MSQL_XP` wait type, chances are that extended stored procedures are being used and are taking longer than normal to complete. Your first point of action should be to identify which extended stored procedures are being used and what they are being used for. Because extended stored procedures can also perform tasks outside SQL Server, they can run into other Windows processes that can slow them down. Knowing what the extended stored procedure function is, and what it does, can help you quickly identify where it is running into issues.

If you are using MARS, you are probably running into MARS-connection deadlocks. There have been various SQL Server updates that reduce the chances of MARS deadlocks occurring, so make sure your SQL Server instance is patched. Also make sure to check the application code that executes queries using MARS for potential issues.

MSQL_XP Summary

The MSQL_XP wait type does two different things: it detects the time it takes to execute extended stored procedures and serves as deadlock detection for MARS connections. Seeing higher-than-normal wait times on the MSQL_XP wait type frequently indicates an extended stored procedure is taking longer than normal to complete. Try to detect which extended stored procedure is being executed and what its function is, as this will make troubleshooting the extended stored procedure easier.

OLEDB

The OLEDB wait type occurs whenever SQL Server has to access the Object Linking and Embedding Database (OLEDB) Client Provider. There are various reasons why SQL Server will use the OLEDB Client Provider, and whenever it does SQL Server will record wait time on the OLEDB wait type.

What Is the OLEDB Wait Type?

SQL Server uses the OLEDB Client Provider for many different actions inside SQL Server. For instance, linked server traffic will move through the OLEDB Client Provider and will result in OLEDB waits. Other actions, especially when SQL Server has to retrieve data from an outside source, can also result in OLEDB Client Provider usage.

Some actions inside SQL Server will also use the OLEDB Client Provider, even though they occur internally. One good example of this is the DBCC command, which I will demonstrate in the following example section.

OLEDB Example

One interesting process that uses the OLEDB Client Provider is the DBCC command inside SQL Server. Whenever you execute a DBCC command, you are bound to see OLEDB waits occur. Listing 12-3 shows an example of OLEDB waits occurring after a DBCC CHECKDB. The example script will clear the sys.dm_os_wait_stats DMV, perform a CHECKDB against the AdventureWorks database, and then query the sys.dm_os_wait_stats DMV for OLEDB waits.

Listing 12-3. Generate OLEDB waits

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
DBCC CHECKDB('AdventureWorks');
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'OLEDB';
```

The results of the query in Listing 12-3 as performed against my test SQL Server instance can be seen in Figure 12-8.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	OLEDB	89954	3292	42	0

Figure 12-8. OLEDB waits

As you can see from Figure 12-8, performing a DBCC CHECKDB will lead to OLEDB waits.

Lowering OLEDB Waits

As you could see in the previous example, performing a DBCC CHECKDB against a database will result in OLEDB waits. This doesn't mean there is a problem related to the OLEDB Client Provider, however; rather, it just indicates that the DBCC CHECKDB command makes use of the OLEDB Client Provider. Running DBCC CHECKDB is a vital part of making sure your databases are healthy. Avoiding consistency checks just to lower OLEDB wait times is bad practice, and I strongly advise against it. Seeing high OLEDB wait times occur outside DBCC commands can indicate there is a performance issue somewhere in your SQL Server environment. If you are dealing with remote sources, such as linked servers or Excel files, you are also affected by the performance of the remote source. For instance, if you are querying information from a linked server and the linked server is experiencing performance problems, it will probably also be reflected in the OLEDB wait time. Also, certain operations, like sorts, can also impact the query duration on the linked server. Network connections to the remote source can also play a role in higher-than-normal OLEDB wait times. If the network connection through which you are accessing your remote source experiences performance degradation, you will again notice this in the OLEDB wait times.

Because the OLEDB wait can occur for various reasons, some of which are benign like DBCC commands, and some that can be related to performance issues, I advise you not to ignore the OLEDB wait type, but rather to monitor it like other performance-indicating wait types.

OLEDB Summary

The OLEDB wait type can occur due to various sources that use the Object Linking and Embedding Database (OLEDB) Client Provider. Most of the sources are related to remote data sources, like linked servers. Some internal processes also use the OLEDB Client Provider, most notably the DBCC command. Seeing higher-than-normal wait times on the OLEDB wait type doesn't have to mean there is a performance problem, especially when they can be correlated to a planned DBCC command execution. Seeing higher-than-normal wait times outside DBCC command when you are using remote data sources like linked servers can mean that the remote data source is experiencing performance problems. In this case, focus on the data source; if the source has problems it is bound to affect the OLEDB wait times as well.

TRACEWRITE

The TRACEWRITE wait type is a special wait type that only collects wait time when a trace is running, and most commonly a SQL Profiler trace. A trace is a background process in SQL Server that collects various, often user-specified, information about the performance of a SQL Server instance. For example, it is possible to use SQL Server Profiler to capture currently executing queries, filtered against a single database, with runtime information. There are various trace methods available in SQL Server, but the most common one that affects the TRACEWRITE wait type is the SQL Server Profiler trace.

SQL Server Profiler is an application that is part of the SQL Server, and starting from SQL Server 2016 the separate SQL Server Management Studio product, and allows users to create and monitor traces against SQL Server instances. The SQL Server Profiler was announced as deprecated by Microsoft with the introduction of SQL Server 2012, and Microsoft recommends using Extended Events to capture traces. Even though the SQL Server Profiler is deprecated, it is still available in SQL Server 2014 and is installed whenever you deploy the separate SQL Server Management Studio product from SQL

Server 2016 onward. Many people still rely on SQL Server Profiler traces instead of Extended Events to troubleshoot and monitor query performance.

The bad news about using SQL Server Profiler is that it can cause some performance overhead while a trace is being performed. Microsoft released an article that concluded that running SQL Server Profiler traces on busy systems can have an impact of 10% on the amount of transactions per seconds; you can find the article here: <https://msdn.microsoft.com/en-us/library/cc293614.aspx>. Because SQL Server Profiler traces can have such a big impact on the performance of your system, I believe it is important to monitor the TRACERWRITE wait time.

What Is the TRACERWRITE Wait Type?

As we just noted, the TRACERWRITE wait type will show up on your system when traces are being performed against your SQL Server instance using the SQL Server Profiler. Because SQL Server Profiler traces can have such an impact on the performance of your SQL Server instance, it is advisable that you monitor the wait type to detect if any SQL Server Profiler traces are being performed.

There are a variety of reasons why you would want to run a SQL Server Profiler trace; for instance, if you want to troubleshoot a very specific query problem or when monitoring how many times a specific query gets executed. Even though there are alternatives to the SQL Server Profiler, like server-side traces and Extended Events, the SQL Server Profiler tool is very easy to use compared to the often complex Extended Events.

TRACERWRITE Example

To show you an example of TRACERWRITE waits occurring, we are going to have to start a SQL Server Profiler trace. The SQL Server Profiler program is part of the Management Tools - Complete feature, which you can select when installing SQL Server versions lower than SQL Server 2016, or when adding features to an existing installation. If you install the separate SQL Server Management Studio product, the Profiler feature is automatically installed as well. Figure 12-9 shows the feature inside the SQL Server 2012 Setup.

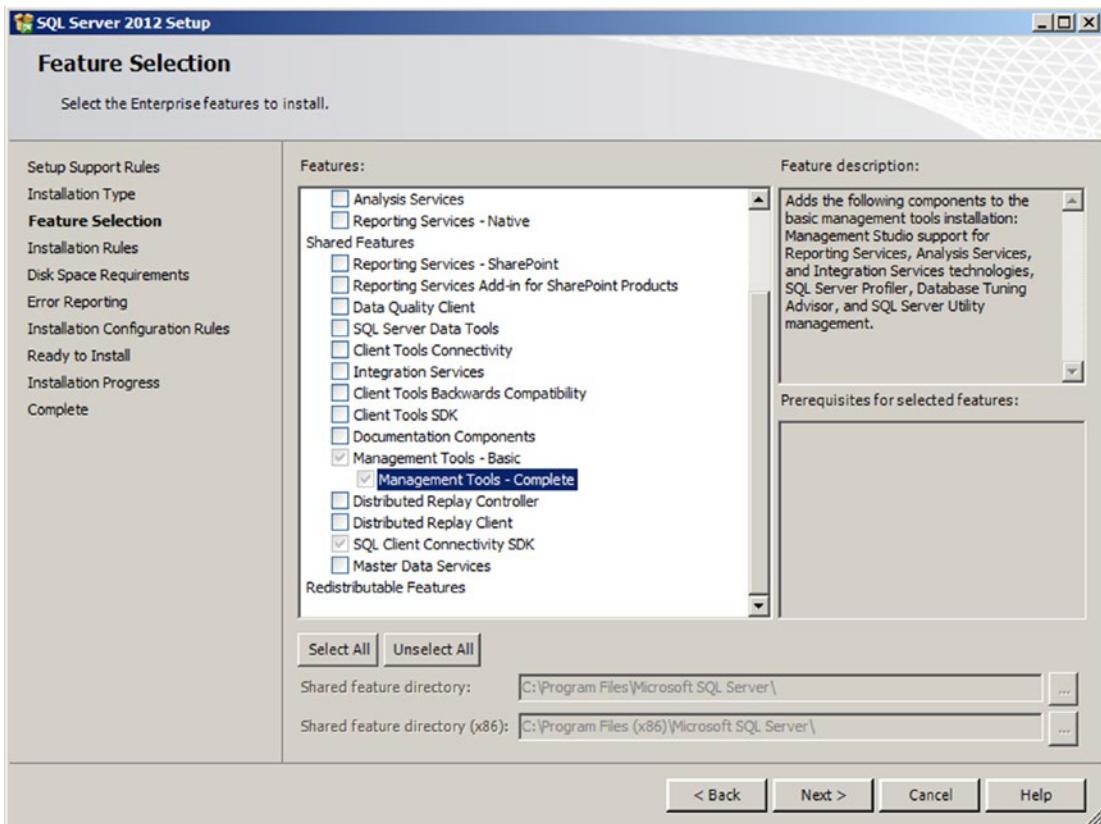


Figure 12-9. Management Tools - Complete feature in SQL Server 2012 Setup

When you have installed the Management Tools - Complete feature, you can find the SQL Server Profiler in the SQL Server ► Performance Tools folder underneath the Start menu, or in the C:\Program Files (x86)\Microsoft SQL Server\[edition number]\Tools\Binn folder.

If you installed the separate SQL Server Management Studio product the Profiler can be found in the C:\Program Files (x86)\Microsoft SQL Server\140\Tools\Binn folder.

After starting up the SQL Server Profiler, you can start a new trace by clicking the New Trace button, shown in Figure 12-10, or selecting File ► New Trace.

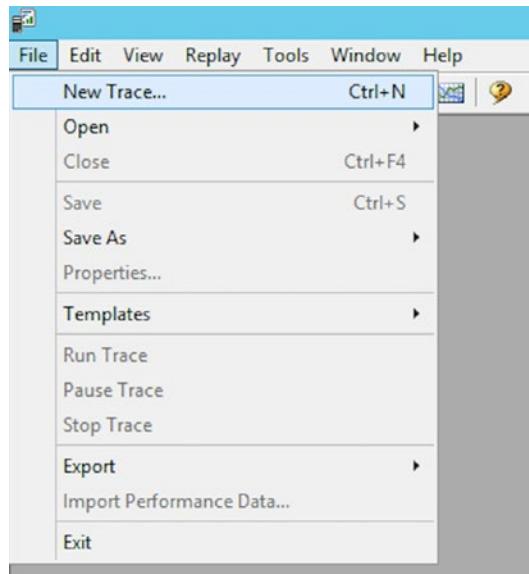


Figure 12-10. New SQL Server Profiler trace

When you start a new trace, you will need to connect to the SQL Server instance you want to trace. In this case I connected to my test SQL Server instance. After logging on to the SQL Server instance, the Trace Properties window will open. This window will give you a variety of options with which to configure your trace and how you want to store your trace. In this example we are not going to change anything in the General tab, but instead will go directly to the Events Selection tab. There we can select what events we want to capture, and optionally supply filters for those events. By default a selection is preloaded when starting a new trace, as you can see in Figure 12-11.

CHAPTER 12 BACKGROUND AND MISCELLANEOUS WAIT TYPES

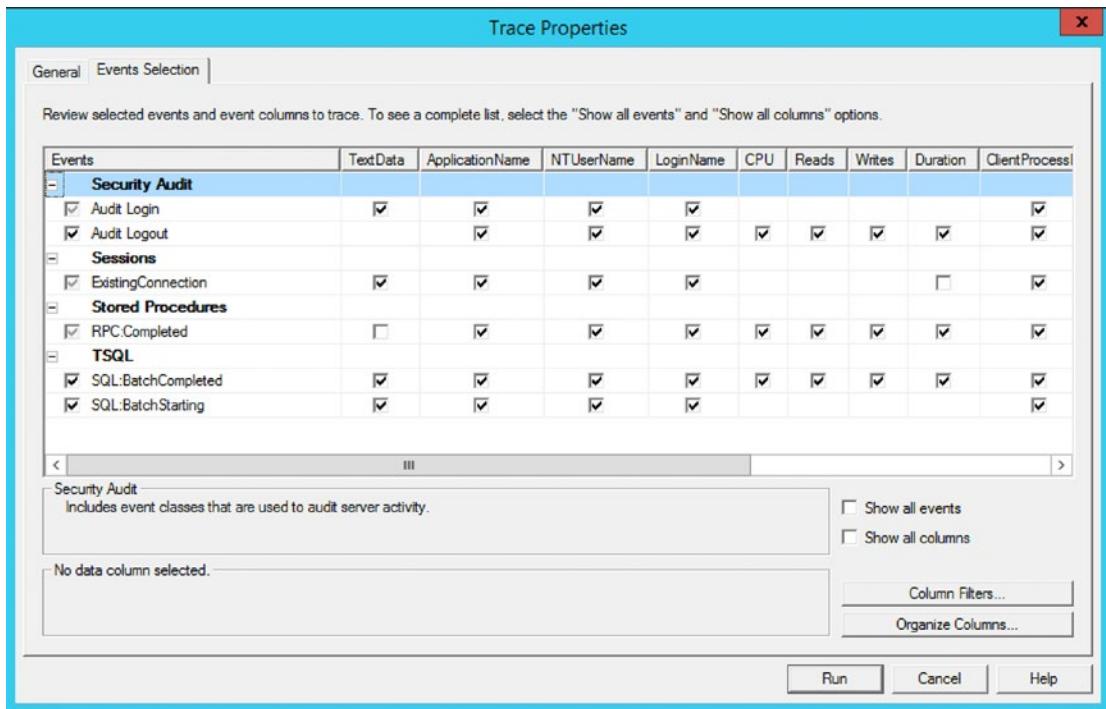


Figure 12-11. SQL Server Profiler default event selection

For this example we do not need all the extra events that are selected by default. In this case we check the Show all columns checkbox, and only select the SQL:BatchCompleted event, as you can see in Figure 12-12. This will record all the T-SQL statements that are executed against the test SQL Server instance and capture all the information available for the event.

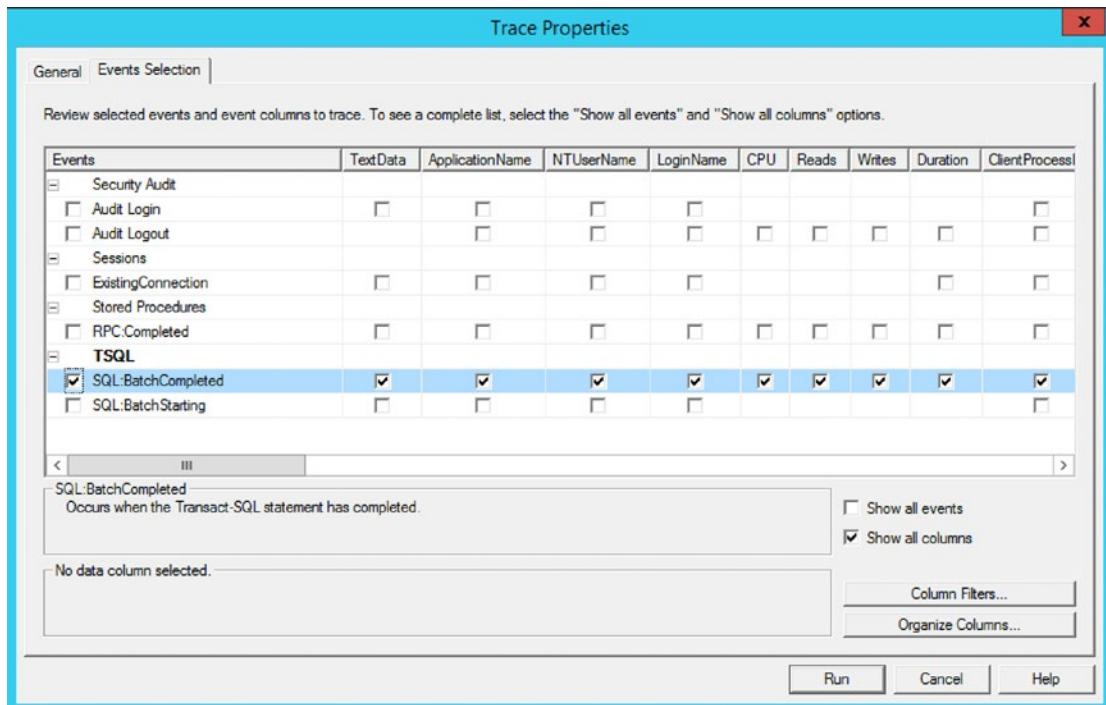


Figure 12-12. *SQL:BatchCompleted* event selected

We won't configure any filters on the event, so we will capture every T-SQL statement we execute against the SQL Server instance. We press Run to start the trace, which will open the trace window that will show us the events when they take place on our SQL Server instance along with additional information about, in this case, the query.

Now that our SQL Server Profiler trace is running, we should be able to notice TRACEWRITE waits occurring. We execute the query that follows in SQL Server Management Studio against the sys.dm_os_waiting_tasks DMV:

```
SELECT *
FROM sys.dm_os_waiting_tasks
WHERE wait_type = 'TRACEWRITE';
```

The results of this query are shown in Figure 12-13.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x00000098F5F5A108	61	0	7	TRACEWRITE	0x0000000000000001

Figure 12-13. TRACEWRITE waits

Even though we are not running any workload on the test SQL Server instance, the TRACEWRITE wait type will still be logged. This is normal since the TRACEWRITE wait type will always be recorded as long as a SQL Server Profiler trace is active.

Lowering TRACEWRITE Waits

As I mentioned before, if you notice TRACEWRITE waits occurring it means someone is running a SQL Server Profiler trace against your SQL Server instance. Because a SQL Server Profiler trace can have such a big impact on the performance of your SQL Server instance, it is important to know who is running the SQL Server Profiler trace and why.

Thankfully, there is a catalog view we can query to view trace activity—the sys.traces view. The sys.traces catalog view will give you an overview of traces that are either active or paused against the SQL Server instance. The query that follows will retrieve all the information inside the sys.traces catalog view:

```
SELECT *
FROM sys.traces;
```

Running this query against the test SQL Server instance returns the information shown in Figure 12-14. (some columns did not fit inside the image).

	id	status	path	max_size	stop_time	max_files	is_rowset	is_rollover
1	1	1	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...	20	NULL	5	0	1
2	2	1	NULL	NULL	NULL	NULL	1	0

Figure 12-14. sys.traces

Some important columns I want to highlight from the sys.traces catalog view are the status and reader_spid columns. The status column returns either a 0 or a 1, where a 0 indicates the trace is stopped or paused and a 1 indicates the trace is currently running. The reader_spid column returns the session ID of the session that started the trace. We can use this information to detect who is running the trace.

In our case, the trace we started in the example has an ID of 2, while the ID of 1 is reserved for the background SQL Server trace that is, by default, always active. This default trace collects specific information about the health of the SQL Server instance and can be used when troubleshooting. Because it is a so-called server-side trace, it does not record TRACEWRITE wait time while it is running.

Now that you can identify the user that is running the trace you can take action if you believe the trace has a negative effect on the performance of your SQL Server instance.

After stopping SQL Server Profiler traces in order to lower the TRACEWRITE wait time, there are other methods available if you really need to capture traces against your SQL Server instance. The most logical one is recreating your SQL Server Profiler trace within an Extended Event session. Extended Events have a much smaller overhead than SQL Server Profiler traces and allow even more events and options while capturing traces.

If you still want to use SQL Server Profiler to analyze traces, it can be a good idea to convert the trace you would normally run in the SQL Server Profiler application to a server-side trace. Just like with Extended Events, server-side traces have minimal overhead compared to traces that are performed through the SQL Server Profiler application. Let's convert the SQL Server Profiler trace we created in the example section to a server-side trace and monitor the effects on the TRACEWRITE wait type.

The easiest way to convert a SQL Server Profiler trace is by defining the trace in the SQL Server Profiler application without starting it. Instead, select the File ► Export ► Script Trace Definition ► For SQL Server 2005 – SQL2017 option, as shown in Figure 12-15.

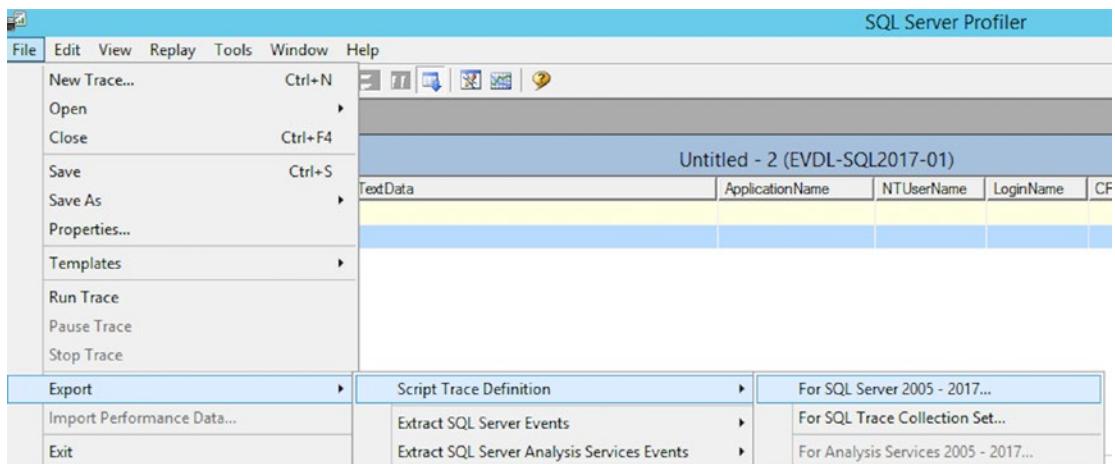


Figure 12-15. Export SQL Server Profiler trace to trace definition

After clicking the File ► Export ► Script Trace Definition ► For SQL Server 2005 – SQL2017 option, we will be asked to save a .sql file. The entire trace definition will be scripted inside this .sql file. We can open this file in SQL Server Management Studio, modify the file location and some other options inside the script, and execute it.

CHAPTER 12 BACKGROUND AND MISCELLANEOUS WAIT TYPES

This will return the ID of the trace we just created and save the trace information to a file we specified at the top of the script. Figure 12-16 shows a part of the exported trace definition on our test SQL Server instance.

```
/*
/* Created by: SQL Server 2017 Profiler      */
/* Date: 03/06/2019  03:00:39 PM            */
*******/

-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e.g., c:\MyFolder\MyTrace. The .trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share

exec @rc = sp_trace_create @TraceID output, 0, N'InsertFileNameHere', @maxfilesize, NULL
if (@rc != 0) goto error

-- Client side File and Table cannot be scripted

-- Set the events
declare @on bit
set @on = 1
```

Figure 12-16. Trace definition

After executing the script to create a server-side trace, we received a trace ID of 2. The trace ID is very important because it is the only way to either start or stop the server-side trace. After creation, the server-side trace is automatically started. If we query the sys.traces catalog view, we can see the server-side trace that was just created, as shown in Figure 12-17.

	id	status	path	max_size	stop_time	max_files	is_rowset	is_rollover
1	1	1	C:\Program Files\Microsoft SQL Server\MSSQL14.MSS...	20	NULL	5	0	1
2	2	1	C:\trace.trc.trc	5	NULL	1	0	0

Figure 12-17. sys.traces

The only way to interact with the server-side trace we created is to execute the `sp_trace_setstatus` stored procedure and supply the trace ID and a status ID. For instance, executing the query that follows will stop the server-side trace with a trace ID of 2:

```
EXEC sp_trace_setstatus 2, 0
```

To start it again we can execute this command:

```
EXEC sp_trace_setstatus 2, 1
```

And finally, to close the trace entirely we can execute the following command:

```
EXEC sp_trace_setstatus 2, 3
```

This does not delete the server-side trace though. As a matter of fact, server-side traces are only removed by a restart of the SQL Server service.

Because a server-side trace can only capture to a trace file, you can navigate to the file you supplied in the server-side trace definition and open the file in SQL Server Profiler. Thus, you can capture the same information as by using the SQL Server Profiler application but at a much lower performance price.

TRACEWRITE Summary

The TRACEWRITE wait type indicates a SQL Server Profiler trace is currently being performed against the SQL Server instance. SQL Server Profiler traces can have a pretty big impact on the performance of your SQL Server instance, and for this reason it is important to monitor the number of traces running against your SQL Server instance. Thankfully, there are some alternatives to SQL Server Profiler traces. You can either choose to convert your SQL Server Profiler trace to an Extended Events session or execute the SQL Server Profiler trace using server-side tracing.

WAITFOR

The final wait type in this chapter is one of the few wait types that are directly related to a T-SQL command. The WAITFOR wait type doesn't indicate performance problems, though it definitely has an impact on the duration of the query that is executing the related WAITFOR T-SQL command.

What Is the WAITFOR Wait Type?

The WAITFOR wait type will get recorded whenever a query is being executed that uses the WAITFOR command. The WAITFOR T-SQL command will stop the execution of the query until a specific amount of time has passed or a specific point in time has been reached. When that happens, the query execution will continue. The WAITFOR command is frequently used inside queries or scripts to force a pause inside the query execution. For instance, in Chapter 4, “Building a Solid Baseline,” we used the WAITFOR command to wait a specific amount of time so we could compare two measurements taken 15 minutes apart.

While pausing the query execution using the WAITFOR command, the transaction holding the WAITFOR command will remain open until the entire transaction has completed. This means that threads are being held by the transaction that cannot be used for other processes. SQL Server also reserves a dedicated thread just for the WAITFOR command; if too many threads are associated with WAITFOR commands and thread starvation occurs, SQL Server will select random WAITFOR threads and terminate them to free up more threads.

In many cases the WAITFOR command is explicitly used by the person who wrote the query or script, and in that sense only impacts that specific query or script; thus, there is no reason to be alarmed when seeing high WAITFOR wait times occur. It just indicates that queries are using the WAITFOR command.

WAITFOR Example

To show you a quick example of the WAITFOR wait type, you can execute the query in Listing 12-4. The query will reset the sys.dm_os_wait_stats DMV, execute a WAITFOR DELAY statement that causes the script execution to wait for 30 seconds, and then query the sys.dm_os_wait_stats DMV for WAITFOR waits.

Listing 12-4. WAITFOR waits

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);  
WAITFOR DELAY '00:00:30';
```

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'WAITFOR';
```

When the query in Listing 12-4 finishes, you should see that one WAITFOR wait occurred, having a total wait time of roughly 30 seconds, as you can see in Figure 12-18.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WAITFOR	1	30002	30002	0

Figure 12-18. WAITFOR wait

WAITFOR Summary

The WAITFOR wait type is one of the few wait types that are directly related to the execution of a T-SQL command, in this case WAITFOR. The WAITFOR wait type doesn't indicate any performance problems with your SQL Server instance, it just indicates the WAITFOR command is being used by a query or script. The WAITFOR T-SQL command will only impact the execution time of the query or script that uses it; therefore, the only way to lower WAITFOR wait times is by removing the WAITFOR command inside queries.

CHAPTER 13

In-Memory OLTP–Related Wait Types

With the release of SQL Server 2014, Microsoft introduced a brand new SQL Server feature called In-Memory OLTP (or codename Hekaton). In-Memory OLTP is a memory-optimized database engine that is directly integrated into the SQL Server 2014 SQL Server engine. In-Memory OLTP is an enterprise-only feature designed to improve performance—up to 20 times, according to Microsoft—by placing tables entirely into the memory of your SQL Server instance. These memory-optimized tables are fully durable and use lock-and-latch free structures to optimize concurrency control.

With the introduction of In-Memory OLTP, various new wait types have been added to SQL Server 2014. Most of these are recognizable by the _XTP_ (or eXtreme Transaction Processing) section in the wait type name. In this chapter we will take a look at some of these new, In-Memory OLTP-related wait types available in SQL Server 2014 or higher.

Before we dive into the wait types though, let's first take a (simplified and short) look at what In-Memory OLTP is and how it works. I will focus on memory-optimized tables in this chapter. In-Memory OLTP also introduced other features, like natively compiled stored procedures and hash indexes, but these are beyond the scope of this chapter.

Introduction to In-Memory OLTP

The main difference between traditional, disk-based tables and memory-optimized tables is that memory-optimized tables reside completely in the memory of your SQL Server instance. Unlike traditional tables, where data pages from that table are moved from disk into memory and back out again, memory-optimized tables are moved to your system's memory at SQL Server startup and never leave the memory (unless the memory-optimized table is removed, of course). While this might sound a bit scary at

first, memory-optimized tables are, by default, fully durable. This means that if your SQL Server instance crashes, memory-optimized table data is not lost. Of course, having an entire table reside in the memory of your SQL Server instance also has its disadvantages. You need to make sure you have enough free memory to accommodate the entire memory-optimized table (and some extra memory to accommodate row versions used when accessing such tables). Calculating the memory requirements can be difficult, but the following article can help you out <https://msdn.microsoft.com/en-us/library/dn282389.aspx>.

The memory you reserve for memory-optimized tables is claimed by SQL Server and will not be wiped out; if your memory-optimized tables use too much memory, your SQL Server instance will run into memory starvation issues that cause performance degradation or, worst-case, cause SQL Server to crash. This is a major difference compared to, for instance, the buffer cache, where pages are wiped out of memory when memory pressure occurs. Another disadvantage is that many data types or SQL Server features are not supported for memory-optimized tables. The complete list of what can and cannot be used can be found at [https://msdn.microsoft.com/en-us/library/dn246937\(v=sql.120\).aspx](https://msdn.microsoft.com/en-us/library/dn246937(v=sql.120).aspx) and at [https://msdn.microsoft.com/en-us/library/dn133181\(v=sql.120\).aspx](https://msdn.microsoft.com/en-us/library/dn133181(v=sql.120).aspx). With the release of SQL Server 2016, some of the major limitations were resolved, making the feature more attractive and less restrictive.

So, how do memory-optimized tables work, and why do they perform that much faster than traditional disk-based tables? Let's take a look at some of the internals of In-Memory OLTP.

CFPs

Like I mentioned earlier, by default memory-optimized tables are durable (you can choose to create a non-durable table that has its contents cleared on a SQL Server Service restart, but you have to explicitly specify this). The way this durability is achieved is through so-called checkpoint file pairs (CFPs). CFPs consist of two files, a data and a delta file, that exist inside a special memory-optimized filegroup that you have to create for the database where you want to use memory-optimized tables.

Unlike traditional tables that store row data inside data pages, data files store the rows of *all* your memory-optimized tables. I emphasize the word *all* because a single data file can hold the rows of many memory-optimized tables, unlike data pages that

store row data for a single table. The rows inside a data file are stored sequentially based on the time they were inserted into a memory-optimized table. This is different than with data pages, which hold row information inside extents for traditional tables. Because rows are stored sequentially inside the data files, there is a performance increase when reading rows since it eliminates the random reads that occur when reading rows from traditional tables. Figure 13-1 shows an abstract view of a data file and the row data it holds.

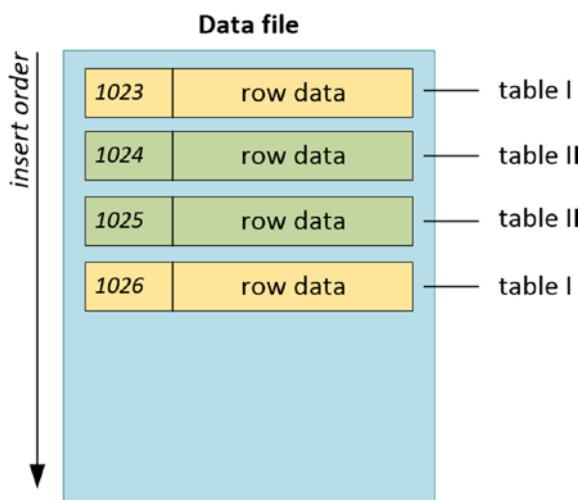


Figure 13-1. Memory-optimized table's data file

There are always multiple data files inside a memory-optimized filegroup. When you first create the memory-optimized filegroup, SQL Server will automatically pre-allocate a number of data files in the file location of the memory-optimized filegroup. The data files will always have a fixed file size, either 128 MB on systems with more than 16 GB memory or 16 MB when there is less than or equal to 16 GB memory. When a data file is full, a new data file will automatically be created and new rows will be inserted into the new data file. It is important to know that the data file keeps track of rows based on the transaction-commit timestamp that inserted the row into the data file (shown by the number inside Figure 13-1). Even if new data files are added and rows are spread across multiple data files, the data files will always have a contiguous range of transactions. Figure 13-2 shows multiple data files and the transaction-commit timestamps associated with those data files.

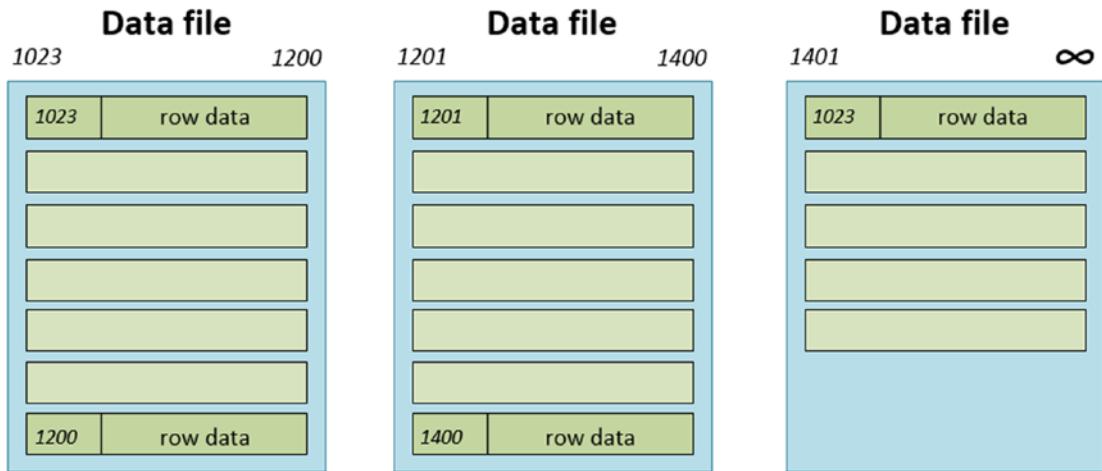


Figure 13-2. Data files and transaction timestamps

Notice in Figure 13-2 that the last data file still has room for new rows—it doesn't have a transaction timestamp to indicate the file is full, so new rows will be added to that file.

Another important characteristic of the data file is that rows that are deleted are not directly removed from the file. Instead they are tracked by the delta file that is associated with the data file. The delta file logs any deletes made in the data file and is connected to the data file by the transaction timestamp range. Row updates for memory-optimized tables are tracked as a delete and insert operation.

The population of the data and delta files is performed by a background thread—called the offline checkpoint thread—that runs constantly in the background of SQL Server. This is different than the checkpoint process used for traditional tables where pages are written to the database data files at intervals. The offline checkpoint thread monitors the transaction log for operations performed on memory-optimized tables and directly writes to the data and delta files.

Over time, when data files accumulate more deleted rows, a merge operation will take place that will merge multiple data files together into one data file. The merge operation will create new data and delta files and move the contents of one or more data and delta files into the new files, but it will not move the rows that were marked as deleted. The transaction-commit timestamps will be adjusted in the new data and delta files so they match the timestamps of the files that were merged. Figure 13-3 shows a simplified view of a merge operation on a data-file level. Keep in mind that a merge will also impact the delta file.

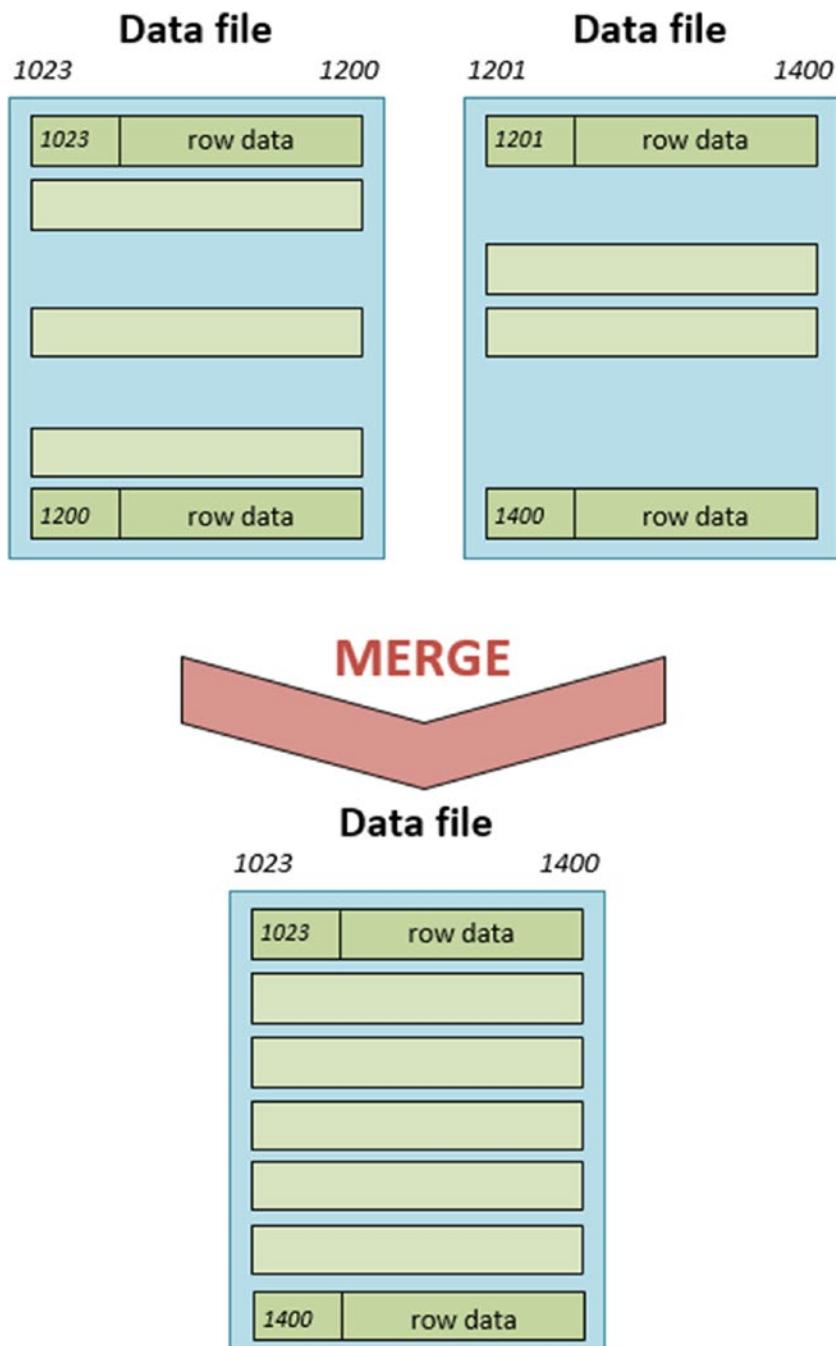


Figure 13-3. Merge operation

Isolation

Concurrent access to memory-optimized tables is handled through snapshot-based transaction isolation. This isolation level shares many characteristics with the snapshot isolation we can use on disk-based tables, but there are some differences so as to optimize throughput. First of all, the snapshot-based transaction isolation uses row versions when concurrent transactions want to access the same row. Instead of storing the row versions in the TempDB database like regular snapshot isolation, the row versions for memory-optimized tables are stored in-line in the data files itself.

Another difference is that snapshot-based transaction isolation uses an optimistic concurrency control. This means that SQL Server assumes no transaction conflict will occur when concurrent transactions access the same data. Because of this assumption there is no need for locks or latches to protect the memory-optimized table data. There is a form of conflict detection active, however, and when it detects that a conflict has occurred, it will end one of the transactions, and that transaction will need to be retried.

Not having to place and maintain locks and latches is another major contribution to the performance of In-Memory OLTP.

Transaction Log Changes

The final differences I want to discuss in this section are the modifications to the behavior of the transaction log regarding memory-optimized tables. For traditional tables, a log record will be generated when a transaction starts whether it gets committed or not. For memory-optimized tables, the log record will only be generated when the transaction begins the commit processing. This means no information for transactions that are rolled back is recorded. This minimizes interaction with the transaction log on disk, thus improving performance.

Another modification is that changes to indexes on memory-optimized tables are not logged in the transaction log. Since indexes that are created on memory-optimized tables are also maintained entirely in-memory, there is no need to record changes. Indexes on memory-optimized tables are regenerated on the start of the SQL Server Service.

The final difference I want to mention is the grouping of multiple transactions into one log record. For traditional tables every transaction will result in at least one log record. Transactions against memory-optimized tables are grouped together and then written as one log record (with a current maximum size of 24 KB). For instance, if you have 200 inserts against a traditional table, at least 200 log records would be generated. If we could fit 100 inserts into one log record for the memory-optimized table, we would only have two log records instead of at least 200. Again, this improves throughput for memory-optimized tables.

Now that we have taken a (simplified and short) look at some of the inner workings of memory-optimized tables, let's move on to some of the wait types that are related to In-Memory OLTP. Most of the three wait types that we will discuss in this chapter are related, one way or another, to the new offline checkpoint process introduced with In-Memory OLTP.

WAIT_XTP_HOST_WAIT

The first wait type we will discuss in this chapter is `WAIT_XTP_HOST_WAIT`. This wait type shares some characteristics with the `CHECKPOINT_QUEUE` wait type we discussed in Chapter 12, “Background and Miscellaneous Wait Types,” in that it seems to be running continuously but only writes its wait information to `sys.dm_os_wait_stats` at specific conditions.

What Is the WAIT_XTP_HOST_WAIT Wait Type?

If we look up some information about the `WAIT_XTP_HOST_WAIT` wait type on Books Online, we get a not-so-helpful definition: “Occurs when waits are triggered by the database engine and implemented by the host.” This doesn’t give us a lot of clues about the processes that might be related to the `WAIT_XTP_HOST_WAIT` wait type, which means we have to do a little bit of digging ourselves.

Before we can start investigating the `WAIT_XTP_HOST_WAIT` wait type, we need to create a memory-optimized table. I used the script shown in Listing 13-1 to create a new database with a single memory-optimized table. There are some path references in this script that you will need to change to make sure the database data and log files are created in the right location.

Listing 13-1. Create test database and memory-optimized table

```
-- Create database
-- Make sure to change the file locations if needed
USE [master]
GO

CREATE DATABASE [OLTP_Test] CONTAINMENT = NONE
ON PRIMARY
(
    NAME = N'OLTP_Test', FILENAME = N'E:\Data\OLTP_Test_Data.mdf' ,
    SIZE = 51200KB , FILEGROWTH = 10%
)
LOG ON
(
    NAME = N'OLTP_Test_log', FILENAME = N'E:\Log\OLTP_Test_Log.ldf' ,
    SIZE = 10240KB , FILEGROWTH = 10%
);
GO

-- Add the Memory-Optimized Filegroup
ALTER DATABASE OLTP_Test ADD FILEGROUP OLTP_MO CONTAINS MEMORY_OPTIMIZED_
DATA;
GO

-- Add a file to the newly created Filegroup.
-- Change drive/folder location if needed.
ALTER DATABASE OLTP_Test ADD FILE (name='OLTP_mo_01', filename='E:\data\
OLTP_Test_mo_01.ndf') TO FILEGROUP OLTP_MO;
GO

-- Create our test table
USE [OLTP_Test]
GO
```

```

CREATE TABLE OLTP
(
    ID INT IDENTITY (1,1) PRIMARY KEY NONCLUSTERED,
    RandomData1 VARCHAR(50),
    RandomData2 VARCHAR(50),
    ID2 UNIQUEIDENTIFIER
)
WITH (MEMORY_OPTIMIZED=ON);
GO

```

Now that we have a memory-optimized table we can use for testing, let's take a look at the `WAIT_XTP_HOST_WAIT` wait type in the `sys.dm_os_wait_stats` and `sys.dm_os_waiting_tasks` DMV using the following query:

```

SELECT *
FROM sys.dm_os_waiting_tasks
WHERE wait_type = 'WAIT_XTP_HOST_WAIT';

SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'WAIT_XTP_HOST_WAIT';

```

The results of this query on my test SQL Server instance can be seen in Figure 13-4.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x000000FD22027848	1	0	18700	WAIT_XTP_HOST_WAIT	NULL

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WAIT_XTP_HOST_WAIT	7	19300626	19300256	0

Figure 13-4. `WAIT_XTP_HOST_WAIT` waits

The first thing you'll notice is that the `WAIT_XTP_HOST_WAIT` wait type is constantly showing up in the `sys.dm_os_waiting_tasks` DMV, and over time the wait time increases. Also, the `session_id` that is related to the `WAIT_XTP_HOST_WAIT` wait type indicates it is an internal SQL Server thread that is recording the wait. From this information we can already formulate some conclusions about the `WAIT_XTP_HOST_WAIT` wait type: it is related to an internal background process that continuously runs. What's

also interesting is that if you were to run the query a second time, the wait time in the `sys.dm_os_waiting_tasks` DMV increases but the wait time in the `sys.dm_os_wait_stats` remains the same. So far we have run into one other wait type that shared this characteristic, the `CHECKPOINT_QUEUE` wait time, which we discussed in Chapter 12, “Background and Miscellaneous Wait Types.”

Since the `CHECKPOINT_QUEUE` wait type has interesting behavior in that it only writes the accumulated wait time of the `sys.dm_os_waiting_tasks` DMV to the `sys.dm_os_wait_stats` DMV when an automatic checkpoint occurs, I decide to simply run a checkpoint command against the `OLTP_Test` database I created using Listing 13-1, and then query both the DMVs again. The impact on the wait times of the `WAIT_XTP_HOST_WAIT` wait type can be seen in Figure 13-5.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
1	0x000000FD22027848	1	0	2611	WAIT_XTP_HOST_WAIT	NULL

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WAIT_XTP_HOST_WAIT	8	19437391	19300256	1

Figure 13-5. `WAIT_XTP_HOST_WAIT` wait information after checkpoint

As you can see in Figure 13-5, the wait time in the `sys.dm_os_waiting_tasks` DMV is very small again, but the wait time in the `sys.dm_os_wait_stats` DMV has increased a lot. Because of this behavior, I believe the `WAIT_XTP_HOST_WAIT` wait type has something to do with the offline checkpoint process that is related to memory-optimized tables.

To verify my guess I need to dig a little bit deeper to find out what goes on underneath the hood of SQL Server when a checkpoint is performed against a memory-optimized table. To get this information I create an Extended Events session that captures the call stack whenever SQL Server runs into a `WAIT_XTP_HOST_WAIT` wait. I won’t bore you with the methods I use for creating this Extended Events session here, but Paul Randal wrote an amazing blog post about capturing call stacks whenever a specific wait occurs that you can use to collect some call stacks yourself. You can find Paul’s blog post here: www.sqlskills.com/blogs/paul/determine-causes-particular-wait-type/.

The results of my Extended Events session for capturing the call stack when a WAIT_XTP_HOST_WAIT wait occurs is shown here:

```
sqldk.dll!XeSosPkg::wait_info::Publish+0x138
sqldk.dll!SOS_Scheduler::UpdateWaitTimeStats+0x2bc
sqldk.dll!SOS_Task::PostWait+0x9e
sqlmin.dll!EventInternal<SuspendQueueSLock>::Wait+0x1fb
sqlmin.dll!HkHostWait::Wait+0xce
hkengine.dll!CkptFilePair::CreateInstance+0x61b
sqlmin.dll!HkHostReportFailure::KillProcess+0x372
sqldk.dll!SOS_Task::Param::Execute+0x21e
sqldk.dll!SOS_Scheduler::RunTask+0xa8
sqldk.dll!SOS_Scheduler::ProcessTasks+0x279
sqldk.dll!SchedulerManager::WorkerEntryPoint+0x24c
sqldk.dll!SystemThread::RunWorker+0x8f
sqldk.dll!SystemThreadDispatcher::ProcessWorker+0x3ab
sqldk.dll!SchedulerManager::ThreadEntryPoint+0x226
kernel32.dll!BaseThreadInitThunk+0xd
ntdll.dll!RtlUserThreadStart+0x21
```

The first part I found really interesting is the inclusion of a new .dll file, hkengine.dll. Since In-Memory OLTP's codename was Hekaton, I am guessing that this .dll holds the new In-Memory OLTP functions, so let's zoom in on that particular call:

```
hkengine.dll!CkptFilePair::CreateInstance+0x61b
```

Seeing the function name, I am guessing it is related to checkpoint file pairs, and the CreateInstance bit suggests a new CFP was created when I executed the CHECKPOINT command. We can verify this by going to the location of the In-Memory file that we created in Listing 13-1. The interesting thing about adding a file to an In-Memory filegroup is that it will actually create a directory, and inside this directory there is a folder with a unique ID string. If you go further down the directory tree, you will end up in a folder with numbered files. Figure 13-6 shows a part of the contents of this folder on my test machine.

Name	Date modified	Type	Size
00000020-000000b0-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000b5-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000c0-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000c5-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000cc-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000d1-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000d8-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000dd-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000e4-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000e9-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000f0-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000000f5-0003	6/26/2015 8:50 AM	File	16,384 KB
00000020-000000fc-0002	6/26/2015 8:50 AM	File	1,024 KB
00000020-000001b0-0002	6/26/2015 9:05 AM	File	16,384 KB
00000020-000001ba-0002	6/26/2015 9:05 AM	File	1,024 KB
00000020-000001bf-0003	6/26/2015 9:06 AM	File	16,384 KB
00000020-000001c6-0002	6/26/2015 9:05 AM	File	1,024 KB

Figure 13-6. In-Memory filegroup files

As a matter of fact, the files you are seeing here are the data and delta files that are associated with the memory-optimized table we created earlier. The 1 MB files are the delta files and the 16 MB ones are the data files.

Since I am guessing a checkpoint would create another CFP, I checked the number of files in the folder before executing a CHECKPOINT, which was 28 files. I then executed a CHECKPOINT command and looked at the number of files again, and it turned out there were now 30 files after the checkpoint.

WAIT_XTP_HOST_WAIT Summary

I believe the WAIT_XTP_HOST_WAIT wait type has a clear relation to the creation of new checkpoint file pairs. Apparently, running a manual CHECKPOINT statement will generate a new CFP for the memory-optimized tables. Because the WAIT_XTP_HOST_WAIT wait type generates wait time constantly in the background, and writes it to the sys.dm_os_wait_stats DMV a new CFP was created (either by manual checkpoint, when an existing CFP was full, or when a Merge operation occurred), I believe the

`WAIT_XTP_HOST_WAIT` wait type does not directly indicate performance problems. It mostly indicates that a new CFP has been added to the In-Memory filegroup. This does not mean this is the only process that generates `WAIT_XTP_HOST_WAIT` waits, though. There can be other processes that can also cause the waits, but so far they only occurred whenever a new CFP needed to be added.

WAIT_XTP_CKPT_CLOSE

The `WAIT_XTP_CKPT_CLOSE` wait type is another new wait type introduced in SQL Server 2014. As the name suggests, it seems to be related to the new offline checkpoint process introduced with the In-Memory OLTP feature.

What Is the `WAIT_XTP_CKPT_CLOSE` Wait Type?

The `WAIT_XTP_CKPT_CLOSE` wait type seems to be related to the new offline checkpoint process that was introduced in SQL Server 2014 with the release of In-Memory OLTP. As far as I can tell by analyzing the behavior of this wait type, it only records wait time when a checkpoint occurs, no matter if it is an automatic or manual checkpoint. The wait time the `WAIT_XTP_CKPT_CLOSE` wait type represents seems to be the time it takes for the checkpoint operation to complete. We can verify this easily by executing a `CHECKPOINT` command against the database and table we created earlier when we discussed the `WAIT_XTP_HOST_WAIT` wait type. I used the script in Listing 13-2 to clear the `sys.dm_os_wait_stats` DMV, insert a few rows inside the memory-optimized table, perform a `CHECKPOINT` operation, and then query the `sys.dm_os_wait_stats` DMV for `WAIT_XTP_CKPT_CLOSE` wait type information.

Listing 13-2. Generate `WAIT_XTP_CKPT_CLOSE` waits

```
USE [OLTP_Test];
GO

-- Clear sys.dm_os_wait_stats
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);

-- Insert some rows
INSERT INTO OLTP
```

```

(
RandomData1,
RandomData2,
ID2
)
VALUES
(
CONVERT(VARCHAR(50), NEWID()),
CONVERT(VARCHAR(50), NEWID()),
NEWID()
);
GO 1000
-- Perform a CHECKPOINT
CHECKPOINT
-- Query sys.dm_os_wait_stats for WAIT_XTP_CKPT_CLOSE waits
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type = 'WAIT_XTP_CKPT_CLOSE';

```

The results can be seen in Figure 13-7.

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	WAIT_XTP_CKPT_CLOSE	2	896	896	0

Figure 13-7. WAIT_XTP_CKPT_CLOSE waits

Just as I did for the WAIT_XTP_HOST_WAIT wait type, I also captured the call stack when a WAIT_XTP_CKPT_CLOSE wait occurred:

```

sqldk.dll!XeSosPkg::wait_info::Publish+0x138
sqldk.dll!SOS_Scheduler::UpdateWaitTimeStats+0x2bc
sqldk.dll!SOS_Task::PostWait+0x9e
sqlmin.dll!EventInternal<SuspendQueueSLock>::Wait+0x1fb
sqlmin.dll!HkCheckpointCtxtImpl::WaitForCkptComplete+0xd0
sqlmin.dll!HkHostWaitForCkptComplete+0x13a

```

```

sqlmin.dll!CheckpointWithOptionalTruncate+0xe6
sqllang.dll!CStmtCheckpoint::XretExecute+0xe7
sqllang.dll!CMsqlExecContext::ExecuteStmts<1,1>+0x427
sqllang.dll!CMsqlExecContext::FExecute+0xa33
sqllang.dll!CSQLSource::Execute+0x86c
sqllang.dll!process_request+0xa57
sqllang.dll!process_commands+0x4a3
sqldk.dll!SOS_Task::Param::Execute+0x21e
sqldk.dll!SOS_Scheduler::RunTask+0xa8
sqldk.dll!SOS_Scheduler::ProcessTasks+0x279
sqldk.dll!SchedulerManager::WorkerEntryPoint+0x24c
sqldk.dll!SystemThread::RunWorker+0x8f
sqldk.dll!SystemThreadDispatcher::ProcessWorker+0x3ab
sqldk.dll!SchedulerManager::ThreadEntryPoint+0x226
kernel32.dll!BaseThreadInitThunk+0xd
ntdll.dll!RtlUserThreadStart+0x21

```

I believe the most interesting line is `sqlmin.dll!CheckpointWithOptionalTruncate+0xe6`, which seems to be the function that performs the truncate. It is followed by the `sqlmin.dll!HkCheckpointCtxtImpl::WaitForCkptComplete+0xd0` line that I believe records the time the previous checkpoint function took place, which gets posted later on to the wait statistics DMVs.

I don't believe seeing `WAIT_XTP_CKPT_CLOSE` waits occur is a direct cause for concern. They indicate that checkpoints are being performed. I can imagine that sudden high wait times for the `WAIT_XTP_CKPT_CLOSE` wait type can indicate a performance issue. As we saw in the previous section, performing a checkpoint against a memory-optimized table will result in extra CFPs being created. I am guessing that if the allocation of CFPs takes a long time, the checkpoint operation will take longer to complete as well, resulting in higher `WAIT_XTP_CKPT_CLOSE` wait times. The amount of data a checkpoint has to process will probably also mean higher `WAIT_XTP_CKPT_CLOSE` wait times. Since the checkpoint writes data to the storage subsystem, the performance of your storage will probably also impact `WAIT_XTP_CKPT_CLOSE` wait times.

WAIT_XTP_CKPT_CLOSE Summary

The WAIT_XTP_CKPT_CLOSE wait type seems closely related to performing checkpoint operations. It indicates the time a checkpoint performed against a memory-optimized table took to complete. I don't believe this directly indicates performance issues, since it just records the time it took for the checkpoint to complete. The amount of work a checkpoint has to process will probably result in higher WAIT_XTP_CKPT_CLOSE wait times. Storage subsystem performance will probably also impact WAIT_XTP_CKPT_CLOSE wait times.

WAIT_XTP_OFFLINE_CKPT_NEW_LOG

The final In-Memory OLTP related wait type that I want to discuss in this chapter is the WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type. This is another wait type related to the offline checkpoint process that was introduced in SQL Server 2014.

What Is the WAIT_XTP_OFFLINE_CKPT_NEW_LOG Wait Type?

The WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type appears to be a benign wait type that records the length of time the offline checkpoint process is waiting for work. This is confirmed by Books Online, which has the following definition: "occurs when offline checkpoint is waiting for new log records to scan."

As we discussed earlier in this chapter, the offline checkpoint process monitors the transaction log for transactions that impact memory-optimized tables so those transactions can be recorded in the data and delta files. This is a constantly running process in the background of SQL Server, which means you will see an internal process with the WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type when you query the sys.dm_os_waiting_tasks DMV, as you can see in Figure 13-8.

	waiting_task_address	session_id	exec_context_id	wait_duration_ms	wait_type	resource_address
24	0x000000FD19CC7848	27	0	19918605	BROKER_TRANSMITTER	NULL
25	0x000000FD19CC7C28	28	0	905	SLEEP_TASK	NULL
26	0x000000FD16500108	29	0	343	HADR_FILESTREAM_IOMGR_IOCOMPLETION	NULL
27	0x000000FD0ED504E8	44	0	3783	WAIT_XTP_OFFLINE_CKPT_NEW_LOG	NULL
28	0x000000FD1DB424E8	2	0	98	LOGMGR_QUEUE	0x000000FDEB69D650

Figure 13-8. WAIT_XTP_OFFLINE_CKPT_NEW_LOG waits inside sys.dm_os_waiting_tasks

Unlike the WAIT_XTP_HOST_WAIT wait type that only writes its wait time information to the sys.dm_os_wait_stats DMV when specific conditions occur, the WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type appears to wait for around 5 seconds, adds the wait time to the sys.dm_os_wait_stats DMV, and then resets the wait time in the sys.dm_os_waiting_tasks DMV again. This might suggest that the offline checkpoint process checks for new work at an interval of around 5 seconds.

To understand a little bit more about the offline checkpoint process, I captured a stack dump whenever a WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait occurred. The stack dump gives us some interesting insight into the process itself, as you can see here:

```
sqldk.dll!XeSosPkg::wait_info::Publish+0x138
sqldk.dll!SOS_Scheduler::UpdateWaitTimeStats+0x2bc
sqldk.dll!SOS_Task::PostWait+0x9e
sqlmin.dll!EventInternal<SuspendQueueSLock>::Wait+0x1fb
sqlmin.dll!SequencedObject<LogBlockId, SequencedWaitInfo<LogBlockId>,0>::WaitUntilSequenceAdvances+0x160
sqlmin.dll!OfflineCheckpointWorker::GetNextLogBlock+0x10d
sqlmin.dll!OfflineCheckpointWorker::DoWorkInternal+0xf7
sqlmin.dll!OfflineCheckpointWorker::DoWork+0x3aa
sqlmin.dll!OfflineCheckpointWorker::WorkLoop+0x3fc
sqldk.dll!SOS_Task::Param::Execute+0x21e
sqldk.dll!SOS_Scheduler::RunTask+0xa8
sqldk.dll!SOS_Scheduler::ProcessTasks+0x279
sqldk.dll!SchedulerManager::WorkerEntryPoint+0x24c
sqldk.dll!SystemThread::RunWorker+0x8f
sqldk.dll!SystemThreadDispatcher::ProcessWorker+0x3ab
sqldk.dll!SchedulerManager::ThreadEntryPoint+0x226
kernel32.dll!BaseThreadInitThunk+0xd
ntdll.dll!RtlUserThreadStart+0x21
```

The most interesting parts are when the OfflineCheckpointWorker function is being called. For readability, here is the section that involves the OfflineCheckpointWorker function:

```
sqlmin.dll!SequencedObject<LogBlockId, SequencedWaitInfo<LogBlockId>,0>::WaitUntilSequenceAdvances+0x160
sqlmin.dll!OfflineCheckpointWorker::GetNextLogBlock+0x10d
```

```
sqlmin.dll!OfflineCheckpointWorker::DoWorkInternal+0xf7  
sqlmin.dll!OfflineCheckpointWorker::DoWork+0x3aa  
sqlmin.dll!OfflineCheckpointWorker::WorkLoop+0x3fc
```

Seeing this stack dump makes me believe the offline checkpoint process is started, starts looking for work by reading log records from the transaction log (LogBlock), grabs the first LogBlock it needs to process, and loops until all LogBlocks are processed. When that is done, I suspect the offline checkpoint goes to sleep again, waits for around 5 seconds, then wakes up and checks for new log records.

Seeing this behavior makes me believe the WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type is harmless. It just indicates that the offline checkpoint process is waiting for work to arrive.

WAIT_XTP_OFFLINE_CKPT_NEW_LOG Summary

The WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type is related to the offline checkpoint process and indicates that process is waiting for work to arrive. Because the WAIT_XTP_OFFLINE_CKPT_NEW_LOG wait type only indicates that the offline checkpoint process is waiting for work, I believe the wait type doesn't indicate any performance issues and can probably be safely ignored.

APPENDIX I

Example SQL Server Machine Configurations

During the writing of this book, I used a few different test systems to generate the examples that are used. This appendix will describe the configuration of the systems I used during the examples and wait type demonstrations. If I needed to modify the system to demonstrate a specific wait type or situation occurring, this will be included in the text inside the chapter that holds the demonstration.

All my test systems are virtual machines I created inside Oracle VirtualBox, a free-to-use virtualization software product that you can download from www.virtualbox.org/.

Another tool I frequently used during examples is Ostress. Ostress is part of the RML utilities provided to manage your SQL Server's performance. You can download the RML utilities using this link: www.microsoft.com/en-us/download/details.aspx?id=4511.

Default Test Machine

The table that follows shows the virtual machine configuration I used for the majority of the book, except for Chapter 10, “High-Availability and Disaster-Recovery Wait Types,” which discusses high-availability and disaster-recovery wait types.

Configuration	Value
Computer name	EVDL-SQL2017-01
vCPUs	2–4
Architecture	64-bit
Memory	4 GB
Storage	50 GB System Drive C:\ (SSD)25 GB Data Drive D:\ (SSD)
Data drive layout	D:\Data, MDF FilesD:\Log, LDF FilesD:\Backup, Backup files
Operating system	Windows Server 2012R2
SQL Server edition	SQL Server 2017 Enterprise
SQL Server features	Database Engine Services
SQL Server instance name	MSSQLSERVER (Default instance)

HA/DR Test Machines

The tables that follow show the configurations of the virtual machines I used for demonstrating high-availability and disaster-recovery wait types as described in Chapter 10, “High-Availability and Disaster-Recovery Wait Types.”

Configuration	Value
Computer name	EVDL-DC-01
Role	Domain Controller (PROWAITS)
vCPUs	1
Architecture	64-bit
Memory	512 MB
Storage	20 GB System Drive C:\ (SSD)
Operating system	Windows Server 2012R2

(continued)

Configuration	Value
Computer name	EVDL-SQL-AG01
Role	Principal (mirroring)Primary (AlwaysOn)Failover Cluster node
vCPUs	2
Architecture	64-bit
Memory	2 GB
Storage	25 GB System Drive C:\ (SSD)20 GB Data Drive D:\ (SSD)
Data drive layout	D:\Data, MDF FilesD:\Log, LDF FilesD:\Backup, Backup files
Operating system	Windows Server 2012R2
SQL Server edition	SQL Server 2017 Enterprise
SQL Server features	Database Engine Services
SQL Server instance name	MSSQLSERVER (Default instance)
Computer name	EVDL-SQL-AG02
Role	Mirror (mirroring)Secondary (AlwaysOn)Failover Cluster node
vCPUs	2
Architecture	64-bit
Memory	2 GB
Storage	25 GB System Drive C:\ (SSD)20 GB Data Drive D:\ (SSD)
Data drive layout	D:\Data, MDF FilesD:\Log, LDF FilesD:\Backup, Backup files
Operating system	Windows Server 2012R2
SQL Server edition	SQL Server 2017 Enterprise
SQL Server features	Database Engine Services
SQL Server instance name	MSSQLSERVER (Default instance)

APPENDIX II

Spinlocks

Spinlocks are described by Microsoft as “lightweight synchronization primitives.” The description looks a lot like the one used for latches, which are described as “lightweight synchronization objects.” This is no coincidence, as spinlocks and latches have a lot in common and both are used to serialize access to internal data structures. Both latches and spinlocks are used when access to objects needs to be held for a very short amount of time.

While spinlocks and latches have an identical purpose, there is one large difference between them. Whenever you cannot acquire a latch because there is another incompatible latch already in place, for example, your request is forced to wait, and it will leave the processor and get returned to the Waiter List (the request receives the “SUSPENDED” state). It is then forced to wait inside the Waiter List until the latch can get acquired, and then it moves through the Runnable queue until it can finally get back on the processor. Because latches are treated like a resource for query execution, they are closely related to wait statistics. SQL Server even records the time it has been waiting on acquiring different latch types and classes, which we discussed in Chapter 9, “Latch-Related Wait Types.” There is a relatively large overhead associated with latches, because if a latch cannot be obtained immediately, it has to move through the different phases of the scheduler again before the request can acquire its latch and get executed on the processor.

Spinlocks work very differently than latches, because whenever a spinlock has to wait out another spinlock already in place before it can get placed itself, the thread does not have to leave the processor. Instead, a spinlock will “spin” until it can be acquired. Figure AII-01 shows the difference between latches and spinlocks whenever one has to wait before it can get acquired.

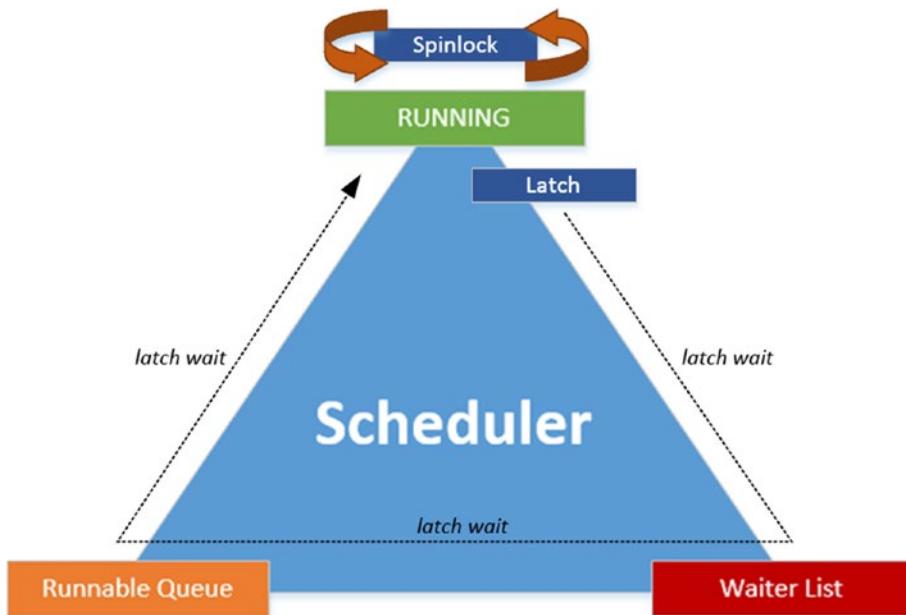


Figure AII-01. Spinlocks and latches and wait phases

The main advantage of using spinlocks instead of latches to synchronize thread access is that spinlocks are even “lighter” synchronization objects than latches. Latches cause extra context switching to occur whenever a latch has to wait before it can get acquired. Spinlocks do not cause context switching because they will never move away from the processor. Because spinlocks do not cause context switching, they are used to protect those areas of SQL Server that are used most intensely. Spinlocks are not the Holy Grail in protecting access to data structures, however. Because they never move away from the processor, they consume processor time, even when they are waiting. To avoid spinlocks consuming too much processor time, every x time around the spinlock will stop spinning and sleep. The interval of the spinlock sleep is calculated by an internal algorithm.

On very busy systems, where many spinlocks are used, it is possible to encounter a phenomenon called spinlock contention. If the spinlock contention gets bad enough, you can notice an increase in processor time that can be difficult to troubleshoot, since this will not always show by analyzing wait statistics.

	name	collisions	spins	spins_per_collision	sleep_time	backoffs
1	SOS_SCHEDULER	64	532294	8317.094	1	8
2	LOCK_HASH	1060	274500	258.9622	0	15
3	RESQUEUE	238	61500	258.4034	0	3
4	BLOCKER_ENUM	112	29500	263.3929	0	2
5	SOS_SUSPEND_QUEUE	25	24621	984.84	0	8
6	BACKUP_CTX	18	6500	361.1111	0	3
7	DBTABLE	2	4000	2000	0	3
8	SESSION_MANAGER	7	1750	250	0	0
9	SOS_TLIST	4	1500	375	0	1
10	LOCK_RW_SECURITY_CACHE	3	750	250	0	0

Figure AII-02. `sys.dm_os_spinlock_stats`

Thankfully, just like latches, there is a spinlock DMV inside SQL Server that tracks the specific spinlock classes (325 in SQL Server 2017), the amount of time a spinlock had to wait before it could get acquired, and the total number of spins that occurred for that spinlock class. We can access this information by querying the `sys.dm_os_spinlock_stats` DMV like the query here:

```
SELECT *
FROM sys.dm_os_spinlock_stats
ORDER BY spins DESC;
```

This returns results like those shown in Figure AII-02.

The columns returned by the `sys.dm_os_spinlock_stats` are described in the following list:

- **name:** Shows the name of the spinlock class.
- **collisions:** Returns the amount of time this spinlock class encountered a wait event because another spinlock was already in place.
- **spins:** When a spinlock has to wait, it performs a spin. The spins column shows the amount of times spins occurred for this specific spinlock class. You can think of a spin as the amount of time the spinlock had to wait before it could get acquired.
- **spins_per_collision:** The average number of spins per collision.

APPENDIX II SPINLOCKS

- `sleep_time`: Time that was spent sleeping for this spinlock class.
- `backoffs`: The number of times a spinlock went to sleep to allow other threads to use the processor.

While all the columns returned by the `sys.dm_os_spinlock_stats` DMV provide valuable information, the `backoffs` column can be the most interesting when you are suspecting a case of spinlock contention. If you notice very high CPU usage and cannot directly correlate the high CPU usage with queries or specific wait types, but the amount of `backoffs` for a specific spinlock class is very high and increasing quickly, you might have a case of spinlock contention occurring.

Spinlock contention is difficult to troubleshoot since it can have a very large number of causes. Also, information about specific spinlock classes is often lacking, increasing the difficulty of troubleshooting spinlock contention. One method that you can use during the analysis of spinlock contention is building a baseline of the `sys.dm_os_spinlock_stats` DMV by capturing the contents of the DMV at a specific interval, like I described in Chapter 4, “Building a Solid Baseline.” This baseline can give you valuable insight into the usage of spinlocks inside your SQL Server instance. Another great tool to diagnose spinlock contention is Extended Events. By using Extended Events you can trace various spinlock-related events, like spinlock backoffs.

To truly analyze why spinlock-class contention is occurring, you will have to dive even deeper by debugging SQL Server memory dumps and looking through the call stack to find what spinlock class is being accessed. Debugging SQL Server memory dumps to identify spinlock contention is beyond the scope of this book and requires a deep knowledge of the inner workings of SQL Server. Thankfully, there is a free Microsoft whitepaper available on spinlock contention that can give you a few pointers for what to do when dealing with spinlock contention. You can get the whitepaper at www.microsoft.com/en-us/download/details.aspx?id=26666.

APPENDIX III

Latch Classes

Latch Class	Books Online Description	Additional Information
ALLOC_CREATE_RINGBUF	Used internally by SQL Server to initialize the synchronization of the creation of an allocation ring buffer.	Used when creating a ring buffer. A ring buffer briefly holds internal event information in memory and is used for diagnostics.
ALLOC_CREATE_FREESPACE_CACHE	Used to initialize the synchronization of internal free space caches for heaps.	Allocates free space for heaps (tables without a clustered index).
ALLOC_CACHE_MANAGER	Used to synchronize internal coherency tests.	
ALLOC_FREESPACE_CACHE	Used to synchronize access to a cache of pages with available space for heaps and binary large objects (BLOBs). Contention on latches of this class can occur when multiple connections try to insert rows into a heap or BLOB at the same time. You can reduce this contention by partitioning the object. Each partition has its own latch. Partitioning will distribute the inserts across multiple latches.	

(continued)

Latch Class	Books Online Description	Additional Information
ALLOC_EXTENT_CACHE	Used to synchronize the access to a cache of extents that contains pages that are not allocated. Contention on latches of this class can occur when multiple connections try to allocate data pages in the same allocation unit at the same time. This contention can be reduced by partitioning the object of which this allocation unit is a part.	
ACCESS_METHODS_DATASET_PARENT	Used to synchronize child dataset access to the parent dataset during parallel operations.	Used together with the ACCESS_METHODS_SCAN_RANGE_GENERATOR latch class during parallel operations to distribute the work among multiple threads.
ACCESS_METHODS_HOBT_FACTORY	Used to synchronize access to an internal hash table.	
ACCESS_METHODS_HOBT	Used to synchronize access to the in-memory representation of a HoBt.	
ACCESS_METHODS_HOBT_COUNT	Used to synchronize access to a HoBt page and row counters.	Used for page and row count deltas for heaps and B-trees.
ACCESS_METHODS_HOBT_VIRTUAL_ROOT	Used to synchronize access to the root page abstraction of an internal B-tree.	Used when accessing metadata regarding the index's root page. See Chapter 8, "Latch-Related Wait Types," for an example.

(continued)

Latch Class	Books Online Description	Additional Information
ACCESS_METHODS_ CACHE_ONLY_HOB T_ALLOC	Used to synchronize worktable access.	Used for synchronizing access to transparent, temporary tables that are created during query execution.
ACCESS_METHODS_ BULK_ALLOC	Used to synchronize access within bulk allocators.	
ACCESS_METHODS_ SCAN_RANGE_ GENERATOR	Used to synchronize access to a range generator during parallel scans.	
ACCESS_METHODS_KEY_ RANGE_GENERATOR	Used to synchronize access to read-ahead operations during key-range parallel scans.	
APPEND_ONLY_ STORAGE_INSERT_ POINT	Used to synchronize inserts in fast append-only storage units.	
APPEND_ONLY_ STORAGE_FIRST_ ALLOC	Used to synchronize the first allocation for an append-only storage unit.	
APPEND_ONLY_ STORAGE_UNIT_ MANAGER	Used for internal data structure access synchronization within the fast append-only storage unit manager.	
APPEND_ONLY_ STORAGE_MANAGER	Used to synchronize shrink operations in the fast append-only storage unit manager.	
BACKUP_RESULT_SET	Used to synchronize parallel backup result sets.	
BACKUP_TAPE_POOL	Used to synchronize backup tape pools.	
BACKUP_LOG_REDO	Used to synchronize backup log redo operations.	

(continued)

Latch Class	Books Online Description	Additional Information
BACKUP_INSTANCE_ID	Used to synchronize the generation of instance IDs for backup performance monitor counters.	
BACKUP_MANAGER	Used to synchronize the internal backup manager.	
BACKUP_MANAGER_DIFFERENTIAL	Used to synchronize differential backup operations with DBCC.	
BACKUP_OPERATION	Used for internal data structure synchronization within a backup operation, such as database, log, or file backup.	
BACKUP_FILE_HANDLE	Used to synchronize file open operations during a restore operation.	
BUFFER	Used to synchronize short-term access to database pages. A buffer latch is required before reading or modifying any database page. Buffer latch contention can indicate several issues, including hot pages and slow I/Os.	Directly related to buffer latches. When seeing higher-than-expected wait times, check if you are running into buffer latch-related contention.
	This latch class covers all possible uses of page latches. sys.dm_os_wait_stats makes a difference between page latch waits that are caused by I/O operations and read and write operations on the page.	
BUFFER_POOL_GROW	Used for internal buffer manager synchronization during buffer pool grow operations.	
DATABASE_CHECKPOINT	Used to serialize checkpoints within a database.	
CLR PROCEDURE_HASHTABLE	Internal use only.	
CLR_UDX_STORE	Internal use only.	

(continued)

Latch Class	Books Online Description	Additional Information
CLR_DATAT_ACCESS	Internal use only.	
CLR_XVAR_PROXY_LIST	Internal use only.	
DBCC_CHECK_ AGGREGATE	Internal use only.	
DBCC_CHECK_ RESULTSET	Internal use only.	
DBCC_CHECK_TABLE	Internal use only.	
DBCC_CHECK_TABLE_INIT	Internal use only.	
DBCC_CHECK_TRACE_LIST	Internal use only.	
DBCC_FILE_CHECK_OBJECT	Internal use only.	
DBCC_PERF	Used to synchronize internal performance monitor counters.	
DBCC_PFS_STATUS	Internal use only.	
DBCC_OBJECT_METADATA	Internal use only.	
DBCC_HASH_DLL	Internal use only.	
EVENTING_CACHE	Internal use only.	
FCB	Used to synchronize access to the file control block.	
FCB_REPLICA	Internal use only.	
FGCB_ALLOC	Use to synchronize access to round-robin allocation information within a filegroup.	

(continued)

Latch Class	Books Online Description	Additional Information
FGCB_ADD_REMOVE	Use to synchronize access to filegroups for ADD and DROP file operations.	Latch is used when adding or removing files inside a filegroup, or when a file grows. Check auto-growth configuration if you are running into contention.
FILEGROUP_MANAGER	Internal use only.	
FILE_MANAGER	Internal use only.	
FILESTREAM_FCB	Internal use only.	
FILESTREAM_FILE_MANAGER	Internal use only.	
FILESTREAM_GHOST_FILES	Internal use only.	
FILESTREAM_DFS_ROOT	Internal use only.	
LOG_MANAGER	Internal use only.	Indicates transaction-log growth because the log could not be cleared or truncated.
FULLTEXT_DOCUMENT_ID	Internal use only.	
FULLTEXT_DOCUMENT_ID_TRANSACTION	Internal use only.	
FULLTEXT_DOCUMENT_ID_NOTIFY	Internal use only.	
FULLTEXT_LOGS	Internal use only.	
FULLTEXT_CRAWL_LOG	Internal use only.	
FULLTEXT_ADMIN	Internal use only.	
FULLTEXT_AMADMIN_COMMAND_CACHE	Internal use only.	

(continued)

Latch Class	Books Online Description	Additional Information
FULLTEXT_LANGUAGE_TABLE	Internal use only.	
LIST		
FULLTEXT_CRAWL_DM_LIST	Internal use only.	
FULLTEXT_CRAWL_CATALOG	Internal use only.	
FULLTEXT_FILE_MANAGER	Internal use only.	
DATABASE_MIRRORING_REDO	Internal use only.	
DATABASE_MIRRORING_SERVER	Internal use only.	
DATABASE_MIRRORING_CONNECTION	Internal use only.	Responsible for controlling the message flow between database mirrors.
DATABASE_MIRRORING_STREAM	Internal use only.	
QUERY_OPTIMIZER_VD_MANAGER	Internal use only.	
QUERY_OPTIMIZER_ID_MANAGER	Internal use only.	
QUERY_OPTIMIZER_VIEW REP	Internal use only.	
RECOVERY_BAD_PAGE_TABLE	Internal use only.	
RECOVERY_MANAGER	Internal use only.	
SECURITY_OPERATION_RULE_TABLE	Internal use only.	
SECURITY_OBJPERM_CACHE	Internal use only.	

APPENDIX III LATCH CLASSES

Latch Class	Books Online Description	Additional Information
SECURITY_CRYPTO	Internal use only.	
SECURITY_KEY_RING	Internal use only.	
SECURITY_KEY_LIST	Internal use only.	
SERVICE_BROKER_ CONNECTION_RECEIVE	Internal use only.	
SERVICE_BROKER_ TRANSMISSION	Internal use only.	
SERVICE_BROKER_ TRANSMISSION_ UPDATE	Internal use only.	
SERVICE_BROKER_ TRANSMISSION_STATE	Internal use only.	
SERVICE_BROKER_ TRANSMISSION_ERRORS	Internal use only.	
SSBXmitWork	Internal use only.	
SERVICE_BROKER_ MESSAGE_ TRANSMISSION	Internal use only.	
SERVICE_BROKER_ MAP_MANAGER	Internal use only.	
SERVICE_BROKER_ HOST_NAME	Internal use only.	
SERVICE_BROKER_ READ_CACHE	Internal use only.	
SERVICE_BROKER_ WAITFOR_MANAGER	Internal use only.	
SERVICE_BROKER_ WAITFOR_ TRANSACTION_DATA	Internal use only.	

(continued)

Latch Class	Books Online Description	Additional Information
SERVICE_BROKER_	Internal use only.	
TRANSMISSION_		
TRANSACTION_DATA		
SERVICE_BROKER_	Internal use only.	
TRANSPORT		
SERVICE_BROKER_	Internal use only.	
MIRROR_ROUTE		
TRACE_ID	Internal use only.	
TRACE_AUDIT_ID	Internal use only.	
TRACE	Internal use only.	
TRACE_CONTROLLER	Internal use only.	Related to SQL Trace. More information about SQL Trace can be found at https://msdn.microsoft.com/en-us/hh245121.aspx .
TRACE_EVENT_QUEUE	Internal use only.	Seeing contention on this latch class can mean too many traces are running at the time.
TRANSACTION_	Internal use only.	
DISTRIBUTED_MARK		
TRANSACTION_	Internal use only.	
OUTCOME		
NESTING_TRANSACTION_	Internal use only.	
READONLY		
NESTING_	Internal use only.	
TRANSACTION_FULL		

(continued)

APPENDIX III LATCH CLASSES

Latch Class	Books Online Description	Additional Information
MSQL_TRANSACTION_	Internal use only.	
MANAGER		
DATABASE_AUTONAME_	Internal use only.	
MANAGER		
UTILITY_DYNAMIC_	Internal use only.	
VECTOR		
UTILITY_SPARSE_	Internal use only.	
BITMAP		
UTILITY_DATABASE_	Internal use only.	
DROP		
UTILITY_DYNAMIC_	Internal use only.	
MANAGER_VIEW		
UTILITY_DEBUG_	Internal use only.	
FILESTREAM		
UTILITY_LOCK_	Internal use only.	
INFORMATION		
VERSIONING_	Internal use only.	
TRANSACTION		
VERSIONING_	Internal use only.	
TRANSACTION_LIST		
VERSIONING_	Internal use only.	
TRANSACTION_CHAIN		
VERSIONING_STATE_	Internal use only.	
VERSIONING_STATE_	Internal use only.	
CHANGE		
KTM_VIRTUAL_CLOCK	Internal use only.	

Index

A

ASYNC_IO_COMPLETION

- AdventureWorks database, 141
- backup operation, 141–142
- disk sec/write perfmon counters, 146
- instant file initialization, 145
- modification, 142
- perform volume maintenance tasks, 144
- SQL Server setup, 143
- storage subsystem, 139
- sys.dm_os_waiting_tasks query, 142
- visual representation, 140

ASYNC_NETWORK_IO

- AdventureWorks database, 148
- graphical representation of, 147
- meaning, 147
- modification, 149
- queries, 149
- task manager network utilization, 150

B

Background processes, 327

BACKUPBUFFER

- additional backup information, 192
- backup/restore operation, 189, 195
- database backup, 191, 194
- definition of, 190
- generating process, 193
- lowering waits, 194

MAXTRANSFERSIZE option, 195

- process, 190
- reader and writer, 191
- results of, 193, 195

BACKUPIO

- ASYNC_IO_COMPLETION, 197
- definition, 195–196
- internals of, 196
- lowering waits, 197
- modified backup query, 196
- sys.dm_os_wait_stats DMV, 196

BACKUPTHREAD

- AdventureWorks database, 199
- lowering waits, 200
- restore operations, 198
- sys.dm_os_waiting_tasks DMV, 200
- threads, 199

Baseline operations

- adjust and measurement of, 77
- CXPACKET, 75
- database, 77
- definition, 76
- pitfalls, 81–82
- process of, 77
- real-time methods, 75
- types and statistics, 79–81
- visualization, 78–79
- wait statistics analysis
 - analysis flowchart, 97
 - comparison graph, 96

INDEX

Baseline operations (*cont.*)

- database, 83
- delta capture method, 86–89
- disk-read latency, 97
- measurements, 82, 99
- PAGEIOLATCH_SH, 94–95
- performance-analysis flowchart, 92
- reset capture method, 85–87
- SQL server agent/schedule
 - measurements, 89–91
- sys.dm_os_waiting_tasks, 93
- sys.dm_os_wait_stats, 84
- table creation, 84
- WaitStats, 91

Buffer latches, 238

C

Checkpoint file pairs (CFPs)

- data and delta file, 358
- memory-optimized table's
 - data file, 357
- merge operation, 359
- traditional tables, 356–357
- transaction timestamps, 358

CHECKPOINT_QUEUE

- automatic checkpoint operation, 331
- checkpoint process, 329
- query results, 330
- sys.dm_os_waiting_tasks, 328
- sys.dm_os_wait_stats, 331

CMEMTHREAD

- EXECUTE (EXEC) command, 154
- memory objects, 151
- mini-dump, 152
- mutex object, 151
- procedure cache, 153
- query procedure cache, 153

- results of, 153

- shared resource, 152

- Common language runtime (CLR), 321

- Cooperative scheduling, 16

- CXCONSUMER wait type, 112

- CXPACKET, 103

- parallelism configuration, 107–111

- SQL Server 2016 SP2 and 2017

- CU3, 112–113

- wait type

- database configuration, 106

- differences, 107

- parallelism configuration, 105

- parallel queries, 104

- parallel thread distribution, 112

- SELECT operation properties, 110

- skewed workloads, 111–112

- threading, 104

D, E, F, G

DBMIRROR_SEND

- AdventureWorks database, 283

- asynchronous mode, 282

- database mirroring monitor, 280, 286

- data modification

- operation, 280–281

- description of, 283

- lowering waits, 285

- Mirror_Test table

- creation, 283

- insert, 284

- principal server, 284

- synchronous mirroring, 280–281

- sys.dm_os_wait_stats, 284

- transaction-log flow, 282

- Delta capture method, 87–89

- DIRTY_PAGE_POLL, 332–334

Dynamic management views (DMVs), 25
vs. detect waits right now, 38
 blocking information
 queries, 38, 40
 results of, 40
 scenarios, 38
 sys.dm_exec_sessions, 40
 sys.dm_os_waiting_tasks, 39
 wait statistics flowchart, 42
 perfmon (wait statistics), 43, 44
 query store, 70–73
 sys.dm_exec_requests, 33–36
 sys.dm_exec_session_wait_stats, 36–38
 sys.dm_os_waiting_tasks, 29–33
 sys.dm_os_wait_stats, 26–29

H

HADR_LOGCAPTURE_WAIT and HADR_WORK_QUEUE, 287–289
HADR_SYNC_COMMIT
 add columns, 295
 AdventureWorks database, 291
 AlwaysOn Availability Group, 291, 294
 AO_Test table creation, 292–293
 dashboard, 294
 lowering waits, 294
 perfmon counters, 296
 primary and secondary mode, 292–293
 synchronous replication
 mode, 290–291, 297
HA/DR test machines, 374

I, J, K

In-memory OLTP
 CFPs, 356–359
 differences, 355

isolation, 360
memory-optimized tables, 355
transaction log changes, 360–361
IO_COMPLETION, 154
 AdventureWorks database, 155
 backup transaction log, 156
 database-related actions, 155
 lowering waits, 157
 NORECOVERY, 156
 sys.dm_os_wait_stats DMV, 155–156
 transaction log backup, 156
IO latches, 239

L

Latches, 235
 compatibility matrix, 237
 LATCH_[xx], 258–266
 modes, 237
 PAGEIOLATCH_SH, 238
 PAGEIOLATCH_[xx], 266–277
 page-latch contention, 241–247
 PAGELATCH_[xx] (*see*
 PAGELATCH_[xx])
 SQL server, 235
 synchronization object, 236
 sys.dm_os_wait_stats DMV, 238–239
 transactions, 236
 waits, 238
LATCH_[xx]
 ACCESS_METHODS_HOBT_
 VIRTUAL_ROOT, 262
 approach, 265
 B-tree
 index structure, 262
 navigation, 263
 cumulative view, 259
 data structures, 258

INDEX

LATCH_xx (*cont.*)

INDEXPROPERTY function, 264
lowering waits, 265
memory area, 259
non-buffer-related latch classes, 266
non-clustered index, 260
Ostress command, 260
resource_description
 column, 260–261
root page splits, 265
SQL Server instance, 261
sys.dm_db_index_physical_stats, 265
test contention table, 260
TRUNCATE command, 264

LAZYWRITER_SLEEP, 335–337

LCK_M_I[xx]

 COMMIT command, 228
 intent locks, 226, 229
 lowering waits, 229
 SELECT statement, 227
 sys.dm_os_waiting_tasks DMV, 228
LCK_M_SCH_S and LCK_M_SCH_M
 lowering waits, 233
ROLLBACK command, 231
schema locks, 230–233
Sch-M and Sch-S locks, 231
SELECT query, 232
sys.dm_os_waiting_tasks, 231–233
sys.dm_tran_locks DMV, 231–232
transaction, 232

LCK_M_S wait type

 COMMIT/ROLLBACK command, 217
 lowering waits, 218
 modification transaction, 216–218
 resource, 219
 SELECT query, 218
 shared locks, 217
 sys.dm_os_waiting_tasks DMV, 218

LCK_M_U

AdventureWorks database, 222
concurrent data modifications, 221
exclusive lock, 223
lock conversion, 221
lowering waits, 223
Ostress utility, 222
transactions, 220
update lock mode, 223
Update (U) mode, 220

LCK_M_X, 223

 COMMIT command, 224
 exclusive lock, 224
 HumanResources.Employee
 table, 225
 lowering waits, 225
 SELECT statement, 224
 sys.dm_os_waiting_tasks DMV, 225
 sys.dm_tran_locks DMV, 224

Locking and blocking mode
 characteristics, 203
 LCK_M_I[xx], 226–229
 LCK_M_S (*see* LCK_M_S wait type)
 LCK_M_SCH_S and LCK_M_
 SCH_M, 230–233
 LCK_M_U, 223–226
 LCK_M_X, 226–229
 modes and compatibility
 concurrent lock situation, 207
 hierarchy, 207, 208
 isolation levels, 208–211
 levels and locking behavior, 210
 lock compatibility, 206
 parentheses, 205
 querying information, 212–217
 read committed, 211
 resource_description column, 213
 sp_WhoIsActive, 214–215

- sql_text output, 215
- sys.dm_os_waiting_tasks DMV, 212
- sys.dm_tran_locks, 212
- transaction, 203–204
- LOGBUFFER and WRITELOG
 - lowering wait, 162
 - sys.dm_os_wait_stats, 162
 - transaction, 157–159
 - trans_demo database, 160–161
- M**
- MSQL_XP
 - deadlock detection, 340
 - execute extended stored procedures, 337
 - lowering waits, 339
 - results of, 338
 - sys.dm_os_wait_stats, 338
- N**
- Non-buffer latches, 239
- Non-preemptive scheduling, 9
- O**
- Object Linking and Embedding Database (OLEDB)
 - DBCC command, 340–342
 - Ostress command, 373
- P**
- PAGEIOLATCH_xx
 - AdventureWorks database, 269
 - buffer cache, 267, 276
 - disk operations, 267
- data page movement, 268, 277
- diagnostic tool, 272
- IO performance script, 273
- lowering waits, 270
- modification, 274
- memory pages, 266
- monitoring storage, 272
- SELECT query, 269
- SQL Server instance, 274
- storage subsystem, 267
- sys.dm_os_wait_stats, 275
- wait time information, 270
- Page-latch contention, 241–247
- PAGELATCH_xx
 - advantage of, 254
 - AdventureWorks database, 249
 - B-trees, 254, 255
 - clustered index and map, 248, 249, 256
 - contention, 253
 - database design class, 248
 - DBCC IND results, 252
 - graphical representation, 247
 - hash partitioning, 254
 - impact latch contention, 253
 - in-memory pages, 258
 - Insert_Test3 table, 256
 - last-page insert contention, 254
 - lowering waits, 252
 - last-page insert contention, 253
 - non-partitioned and partitioned index, 257
 - Ostress command, 250
 - partition function, 255
 - page in-memory, 247
 - query filters, 250
 - rows distribution, 258
 - TempDB database, 248, 252

INDEX

PREEMPTIVE_OS_

AUTHENTICATIONOPS, 316

authentication requests, 321

lowering waits, 319

mixed-mode authentication, 317

query Window, 318

output results of, 319

SQL Server management, 318

Windows login authentication, 317

PREEMPTIVE_OS_ENCRYPTMESSAGE

and PREEMPTIVE_OS_DECRYPTMESSAGE

certificate account selection, 308–309

connection properties, 311

encrypted connection, 305, 312–313

decryption, 305

features view, 306

lowering waits, 312

output results of, 312

self-signed certificate, 307–309

SQL Server instance, 306

PREEMPTIVE_OS_GETPROCADDRESS

entry-point, 321–322

stored procedures, 325

lowering waits, 325

master database selection, 323

output results of, 333–324

xp_getnetname, 324

PREEMPTIVE_OS_WRITEFILEGATHER

database file configuration, 314

file initialization, 316

storage subsystem, 315–316

WriteFileGather function, 313

Preemptive scheduling model, 8, 301

graphical representation, 301

PAL layer interaction, 304

SQL Server/Linux, 302–305

types (*see* PREEMPTIVE_OS_

ENCRYPTMESSAGE and

PREEMPTIVE_OS_

DECRYPTMESSAGE)

WriteFileGather Windows

function, 302

Q

Query store

architecture, 64–65

vs. DMVs, 64

categories, 70

modification, 72

output, 71–72

queries, 71

runtime_stats_interval_id, 72

statistics collection interval, 73

sys.query_store_wait_stats, 70

feature, 63

flight-recorder, 63

wait statistics

built-in-reports, 68

categories, 69

metric-wait time, 68–69

processes, 65–67

types and categories, 66

R

REDO_THREAD_PENDING_

WORK, 297–300

Reset capture method, 86–87

RESOURCE_SEMAPHORE

additional memory, 163–164

AdventureWorks database, 165

lowering wait, 170

- MemoryGrantInfo, 165, 166
- RequiredMemory property, 166
- required memory, 164
- resource semaphore queue, 171
- resource_semaphore.sql, 167
- SELECT operator, 165
- SerialRequiredMemory, 166
- sys.dm_exec_query_resource_semaphore, 169
- sys.dm_os_waiting_tasks DMV, 167
- workspace memory (KB)
- counter, 170
- RESOURCE_SEMAPHORE_QUERY_COMPILE**
- compilation-memory
- resource, 171–172
- CompileMemory property, 176
- contention, 175
- DBCC MEMORYSTATUS**
- command, 174–176
- execution plan properties, 175
- lowering wait, 177
- resource semaphores, 178
- sys.dm_os_waiting_tasks DMV, 176
- S**
- Shared Intent Update (SIU), 227
- SLEEP_BPOOL_FLUSH**
- CHECKPOINT command, 184
 - checkpoint process, 179, 185
 - data modification process, 180
- DBCC SQLPERF command, 184
- generating waits, 182
- lowering waits, 185
- trans_demo database, 183
- types, 181
- SOS_SCHEDULER_YIELD**
- AdventureWorks database, 116
 - CPU queries, 121
 - lowering wait times, 117–122
 - meaning, 114
 - Ostress execution, 120
 - phases and queues, 114
 - processor, 115
 - RUNNING state, 115
 - situations, 117
 - sys.dm_os_wait_stats, 116
- Spinlocks**
- advantage of, 378
 - backoffs, 380
 - latches, 377–378
 - lightweight synchronization objects, 377
 - sys.dm_os_spinlock_stats, 379–380
- SQL server 2005 architecture, 8
- SQL server agent/Schedule**
- measurements, 89–91
- SQL Server architecture, 7
- Sys.dm_exec_requests**, 11, 33
- queries, 35
 - execution plan, 36
 - statement and plan, 35
 - test system, 36
 - results of, 33
 - wait statistics analysis, 34
- Sys.dm_exec_sessions**, 10
- Sys.dm_exec_session_wait_stats**, 36–38
- Sys.dm_os_tasks**, 12
- Sys.dm_os_waiting_tasks**, 29, 93
- column returns, 30
 - queries, 31–33
 - results of, 30
- Sys.dm_os_wait_stats**, 26–29, 240
- Sys.query_store_wait_stats**, 70

T

Test machine, 373
 THREADPOOL wait type, 123
 administrator connection, 130–131
 AdventureWorks database, 126
 CPU usage history graph, 136
 CXPACKET, 133
 formulas, 124
 gaining access, 130, 131
 Ostress tool, 126, 129, 135
 parallelism, 132–135
 processors configuration, 125
 SQL Server instance, 127
 sys.dm_osSchedulers, 127
 sys.dm_os_waiting_tasks, 128–129
 tasks and worker threads, 128
 test machine, 126
 unresponsive, 129
 user connections, 134–136
 worker threads, 123

TRACEWRITE
 event selection, 346
 lowering waits, 348
 management tools, 344
 output results of, 347
 sp_trace_setstatus, 351
 SQL-BatchCompleted selection, 347
 SQL Server Profiler traces, 342–351
 sys.traces, 348, 350
 trace definition, 343, 350

U, V

Update Intent Exclusive (UIX), 227
 User Mode Scheduling (UMS), 9

W, X, Y, Z

WAITFOR, 351–353
 Wait statistics, 3
 baselines, 82–99
 DMVs together, 18
 extended events, 45
 ALTER EVENT SESSION
 command, 54
 configuration, 48
 event filter, 45, 49, 51, 54
 file as rows, 55, 56
 live data tab, 52
 management folder, 47
 results of, 45, 56
 sessions folder, 47, 52
 SQL Server Profiler, 45
 sql_text global field, 50
 sys.dm_xe_map_values, 46
 history of, 4–6
 perfmon, 43–44
 per-query (execution plans), 57–61
 query store, 65–67
 results of, 18
 scheduler view
 few milliseconds, 22
 phases and queues, 19
 request execution time
 calculation, 21
 Runnable queue, 19
 RUNNING phase, 20
 running requests, 21
 wait times and worker
 thread flow, 20
 SQLOS, 6–9
 sys.dm_exec_sql_text, 18

tasks, schedulers and worker threads, 9
cooperative scheduling, 16–17
requests, 11
sessions, 10–11
SQL server, 9
tasks, 12, 13
worker threads, 13–17
thread scheduling, 3

WAIT_XTP_CKPT_CLOSE
call stack, 368
checkpoint operations, 370
offline checkpoint process, 367
results, 368
`sys.dm_os_wait_stats`, 367

WAIT_XTP_HOST_WAIT
.dll file, 365
extended events session, 365
memory-optimized
tables, 366
results of, 363
shares, 361
`sys.dm_os_waiting_tasks`, 363
test database and memory-optimized
table, 362

**WAIT_XTP_OFFLINE_CKPT_NEW_
LOG**, 370–372

Worker threads, 13–15

WRITE_COMPLETION, 186–187