

Inside Microsoft® SQL Server™ 2005: Query Tuning and Optimization by Kalen Delaney

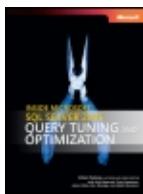
Publisher: Microsoft Press
 Pub Date: September 26, 2007
 Print ISBN-10: 0-7356-2196-9
 Print ISBN-13: 978-0-7356-2196-1
 Pages: 448

[Table of Contents](#)

| [Index](#)

[Overview](#)

Dive deep into the internals of query tuning and optimization in SQL Server 2005 with this comprehensive reference. Understanding the internals of SQL Server helps database developers and administrators to better create, access, and effectively process information from enterprise data. Written by experts on SQL Server, this volume from the Inside Microsoft SQL Server series of books focuses on query tuning and optimization. You'll take an in-depth look at the best ways to make queries more efficient and effective, while maximizing existing resources. Includes extensive code samples and table examples to help database developers and administrators understand the intricacies and help promote mastery of query tuning and optimization.



Inside Microsoft® SQL Server™ 2005: Query Tuning and Optimization by Kalen Delaney

Publisher: Microsoft Press
 Pub Date: September 26, 2007
 Print ISBN-10: 0-7356-2196-9
 Print ISBN-13: 978-0-7356-2196-1
 Pages: 448

[Table of Contents](#)

| [Index](#)

[Copyright](#)

[Foreword](#)

[Acknowledgments](#)

[Introduction](#)

[Chapter 1. A Performance Troubleshooting Methodology](#)

[Factors That Impact Performance](#)

[Troubleshooting Overview](#)

[Summary](#)

[Chapter 2. Tracing and Profiling](#)

[SQL Trace Architecture and Terminology](#)

[Security and Permissions](#)

[Getting Started: Profiler](#)

[Server-Side Tracing and Collection](#)

[Troubleshooting and Analysis with Traces](#)

[Tracing Considerations and Design](#)

[Auditing: SQL Server's Built-in Traces](#)

[Summary](#)

[Chapter 3. Query Execution](#)

[Query Processing and Execution Overview](#)

[Reading Query Plans](#)

[Analyzing Plans](#)

[Summary](#)

[Chapter 4. Troubleshooting Query Performance](#)

[Compilation and Optimization](#)

Detecting Problems in Plans
Monitoring Query Performance
Query Improvements
Query Processing Best Practices
Summary
Chapter 5. Plan Caching and Recompilation
The Plan Cache
Caching Mechanisms
Plan Cache Internals
Objects in Plan Cache: The Big Picture
Multiple Plans in Cache
When to Use Stored Procedures and Other Caching Mechanisms
Troubleshooting Plan Cache Issues
Summary
Chapter 6. Concurrency Problems
New Tools for Troubleshooting Concurrency
Troubleshooting Locking
Troubleshooting Blocking
Troubleshooting Deadlocking
Troubleshooting Row-Versioningâ€”Based Snapshot-Based Isolation Levels
Summary
Additional Resources and References
About the Authors
Kalen Delaney
Sunil Agarwal
Craig Freedman
Adam Machanic
Ron Talmage
Additional Resources for Developers: Advanced Topics and Best Practices
Additional SQL Server Resources for Developers
Prepare for Certification with Self-Paced Training Kits: Official Exam Prep Guidesâ€”Plus Practice Tests
2007 MicrosoftÂ® Office System Resources for Developers and Administrators
Additional Resources for Developers from Microsoft Press
Visual Basic 2005
Visual C# 2005
Web Development
Data Access
SQL Server 2005
Other Developer Topics
More Great Developer Resources from Microsoft Press
Developer Step by Step
Developer Reference
Focused Topics
Index

Copyright

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright Â© 2008 by Kalen Delaney

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2007930321

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 2 1 0 9 8 7

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, Microsoft Press, MSDN, MSN, SQL Server, Visual Studio, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Ben Ryan

Developmental Editor: Devon Musgrave

Project Editor: Maureen Zimmerman

Editorial Production: Carlisle Publishing Services

Technical Reviewers: Benjamin Nevarez and Andy Kelly

Body Part No. X13-86331

Dedication

In fond memory of Jim Gray

Foreword

It has been twenty years since relational database management systems began their transition from research projects and barely usable early product releases into today's primary method of storing and accessing data in commercial applications. Tuning for those early systems consisted largely of choosing the amount of system resources to devote to them, allocating storage, and defining indexes. While this simplicity was often considered a strength, the inability to tune or optimize the application's use of these systems limited their ability to be used in very demanding applications. Today's SQL Server 2005 offers the best of both worlds, continuing to allow many applications to be written with little or no effort applied to tuning and optimization.

while providing extensive tuning and optimization facilities to support the most demanding applications. Whether you are creating a new high-performance SQL Server 2005 application or trying to figure out why an existing application isn't performing as expected, this is the book for you.

This book has its roots in the march from SQL Server 6.5 to SQL Server 7.0. The replacement of SQL Server 6.5's very simplistic Query Processor with a new state-of-the-art design moved Query Processing front and center in any discussion of tuning and optimization. The number of possible execution plans skyrocketed, the importance of allowing the database system to maintain good statistics rose dramatically, and the way queries were compiled and cached suddenly became important to the Database Application Programmer and Database Administrator. New tools for identifying and resolving performance problems were introduced in SQL Server 7.0 and enhanced for SQL Server 2000 and again in SQL Server 2005. New Query Processing-based features such as Indexed Views were added to SQL Server and many existing features, such as the Inserted and Deleted tables in Trigger processing, were migrated to use Query Processing rather than a traditional Storage Engine-based mechanism. With new features in SQL Server 2005 such as Plan Guides and new tools such as Dynamic Management Views, I'm happy to see that Query Tuning and Optimization now has a book of its own.

With the release of SQL Server 2005, Microsoft SQL Server completed its three-release evolution from a purely departmental server to a product capable of handling everything from storage for embedded applications to the most demanding enterprise requirements. Kalen has been a key partner along the way in educating database professionals on how to make the most of Microsoft SQL Server. She has a unique talent for taking material that is typically in the realm of the Computer Science Ph.D. and making it useful for database application professionals. Kalen had worked closely with our team while writing Inside Microsoft SQL Server 7.0, so when I wanted a co-author to turn my TechEd presentation on Query Processor Internals into an MSDN article she was the obvious choice. The latest updates to that material live on in [Chapter 5](#) of this book.

With Query Processing playing such a huge role in database application performance, and with high performance continuing to be a priority for many database applications, this is a book I strongly recommend for all Microsoft SQL Server professionals.

Hal Berenson
Distinguished Engineer
Microsoft Corporation

Acknowledgments

As always, a work like this is not an individual effort, and for this current volume it is truer than ever. I was honored to have four other SQL Server experts join me in writing *Inside SQL Server: Query Tuning and Optimization*, and I truly could not have done this book alone. I am grateful to Sunil Agarwal, Craig Freedman, Adam Machanic, and Ron Talmage for helping to make this book a reality. In addition to my brilliant co-authors, this book could never have seen the light of day with help and encouragement from many other people.

First on my list is you, the readers. Thank you, thank you, thank you for reading what I have written. Thank you to those who have taken the time to write to me about what you thought of the book and what else you want to learn about SQL Server. I wish I could answer every question in detail. I appreciate all your input, even when I'm unable to send you a complete reply. One particular reader of my previous book, *Inside SQL Server 2005: The Storage Engine*, deserves particular thanks. I came to know Ben Nevarez as a very astute reader, who found some uncaught errors and subtle inconsistencies and politely and succinctly reported them to me through my website. After a few dozen e-mails, I started to look forward to Ben's e-mails and was delighted when I got the chance to meet him at the PASS conference in November, 2006. Ben is now one of my technical editors for this current volume, and I am deeply indebted to him for his extremely careful

reading of every one of the chapters.

Thanks to Ron Soukup for writing the first edition of Inside SQL Server and for giving me his "seal of approval" for taking on the subsequent editions. Thanks to my former editor at SQL Server Professional Journal, Karen Watterson, for opening so many doors for me, one of which included passing on my name to the wonderful people at Microsoft Press.

As usual, the SQL Server development team at Microsoft has been awesome. Although Lubor Kollar was not directly involved in much of my query processor research this time, I always knew he was there in spirit, and he always had an encouraging word when I saw him. Sunil Agarwal was as inspirational during the development of this book as he was for the Storage Engine volume, and he was able to help me track down the right people to ask for help with any problems or questions any of the authors had. In addition, Sunil wrote the first chapter for this book, and I am very grateful to him for his hard work.

Stefano Stefani and Sangeetha Shekar met with me and responded to my (sometimes seemingly endless) e-mails. Erik Ismert reviewed a critical piece of my recompilation chapter right at the end and provided valuable information. Eric Hanson provided details of the plan guide internals that I was unable to find anywhere else, and he also provided scripts for working with the plan guide metadata. Paul Randal, Ketan Duvedi, Giri Nair, Sameer Verkhedkar, Milind Joshi, and Andrew Richardson also offered valuable technical insights and information when responding to my e-mails or to questions from Sunil. I hope they all know how much I appreciated every piece of information I received.

I am also indebted to Cindy Gross, Bob Ward, Bob Dorr, Keith Elmore, and Ken Henderson of the SQL Server Product Support team, not just for answering occasional questions, but for making so much information about SQL Server available through white papers conference presentations, and Knowledgebase articles. I am grateful to Alan Brewer, Gail Erickson, and Buck Woody for the great job they and their User Education team did putting together the SQL Server documentation in the Books Online.

I would also like to thank Leona Lowry for finding me office space in the same building as most of the SQL Server team. Thank you, Leona. The welcome you gave me was much appreciated.

My editors at Microsoft Press deserve thanks also. Ben Ryan, my acquisitions editor, got the project off the ground and kept it flying, and overcame many obstacles along the way. Devon Musgrave, my developmental editor, made sure the chapters started coming, and Maureen Zimmerman, the project editor, made sure the chapters kept coming all the way to the end. My external technical editors, Andy Kelly and Ben Nevarez, made sure those chapters turned out well. But, of course, the two Bens, Devon, Maureen, and Andy couldn't do it all by themselves, so I'd like to thank the rest of the editorial team, including manuscript editor Erica Orloff. I know you worked endless hours to help make this book a reality. I would also like to let my agent, Claudette Moore, know how much I appreciate all that she had to put up with from me to get the contract for this book arranged.

I would like to extent my heartfelt thanks to all of the SQL Server MVPs, but most especially Tibor Karaszi and Roy Harvey, who participate in the private newsgroup, for the brilliant exchanges and challenging ideas discussed, and for all the personal support and encouragement given to me. Other MVPs who inspired me during the writing of this volume are Greg Linwood, Hugo Kornelis, Erland Sommarskog, Tony Rogerson, Steve Kass, Tom Moreau, and Linchi Shea. This, of course, includes our former Microsoft MVP lead Ben Miller, and our current lead Steve Dybing. I'd like to say that being a part of the SQL Server MVP team continues to be one of the greatest honors and privileges of my professional life.

I am deeply indebted to my students in my SQL Server Internals classes, not only for their enthusiasm for the SQL Server product, and for what I have to teach and share with them, but for all they have to share with me. Much of what I have learned has been inspired by questions from my curious students. Some of my students, such as Lara Rubbelke, have become friends, and continue to provide ongoing inspiration.

Most important of all, my family continues to provide the rock-solid foundation I need to do the work that I do. My husband, Dan, continues to be the guiding light of life after twenty-two years of marriage. My daughter, Melissa, is a role model for me, and received her Ph.D. in Linguistics from University of Edinburgh about the same time as I was finishing up this book. My three sons, Brendan, Rickey, and Connor are now for the most part all grown, and are all generous, loving, and compassionate young men, still deciding what course their lives will follow for the immediate future. I feel truly blessed to have them in my life.

Kalen Delaney

I would like to thank Kalen Delaney for providing me with this great opportunity to be a part of her book. Though I really enjoyed working on the book, it turned out to be a lot harder than I had originally anticipated but Kalen was always there with her encouraging words to guide me through. I also feel fortunate to have my work reviewed by three of my colleagues, Jerome Halmans, Ron Dar Ziv, and Mike Ruthruff, and other reviewers arranged by the book editors. I do really appreciate all their feedback. Finally, I would like to thank my wife, Anju, and son, Sahil, for their support during the writing of this book and to my parents who have always encouraged me to do my best.

Sunil Agarwal

I would like to express my appreciation to Milind Joshi of the SQL Server team for his time and patience researching and answering so many questions. My contributions to this book are better because of your input as well as that of other members of the SQL Server team. To name but a few: Jay Choe, Campbell Fraser, Lubor Kollar, Martin Neupauer, Shu Scott, Stefano Stefani, and Aleksandras Surna. Last, but certainly not least, thank you Alyssa, Natali, and Gavin for your love and support.

Craig Freedman

First and foremost, I would like to thank Andy Kelly for helping me get involved in this project to begin with. I would also like to extend my thanks to Kalen Delaney, not only for making this whole thing possible, but also for keeping me involved even when I had convinced myself that I wouldn't be able to handle the extra load on my schedule. And finally, I'd like to thank my wife, Kate, who somehow always manages to support me and pick up the slack when I disappear into a writing project.

Adam Machanic

I would like to thank Kalen Delaney for inviting me to contribute a chapter to this book and helping me keep it on track. I also want to thank the Technical Editors for their helpful comments. I am indebted to the numerous articles, white papers, and blogs cited in the Bibliography for examples of troubleshooting concurrency issues. I'd also like to thank my colleagues and customers who, over the years, have provided numerous examples of unusual and challenging deadlocks!

Ron Talmage

Introduction

As I mentioned in the Introduction to Inside SQL Server 2005: The Storage Engine, the most wonderful thing about writing a book and having it published is getting feedback from you, the readers. This continues to hold true as I get feedback from readers of The Storage Engine book. It is very heartening to discover how closely some of you are reading what I have written, and how carefully you are trying to understand every sentence, so that if there are errors, or even parts that are less than crystal clear, you'll discover it and tell me. Although an entire team of editors and reviewers have done their best to make sure there are no errors or unclear concepts, they do occasionally find their way into the published work.

Of course, it's also one of the worst things when I get feedback from readers who are frustrated that their favorite topics were not included. However, by this third time around, I think I can accept the fact that this book cannot be all things to all people, as much as I might want it to be. Microsoft SQL Server 2005 is such a huge, complex product that not even with a new multivolume format can we cover every feature. My hope is that you'll look at the cup as half full instead of half empty and appreciate the volumes of Inside Microsoft SQL Server 2005 for what they do include. As for the topics that aren't included, I hope you'll find the information you need in other sources.

The focus of this SQL Server 2005 series, as the name Inside implies, is on the core SQL Server engine—in particular, the query processor and the storage engine. This series doesn't talk about client programming interfaces, heterogeneous queries, business intelligence, or replication. In fact, most of the high-availability features are not covered, but a few, such as mirroring, are mentioned at a high level when we discuss database property settings. I don't drill into the details of some internal operations, such as security—I had to draw the line somewhere so we wouldn't need 10 volumes in the series and have no hope of finishing it before the release of the next version of the product!

A History of

The first edition of Inside Microsoft SQL Server, written for version 6.5, did attempt to cover almost all features of the product. Back then, the product was much smaller. Also, few other SQL Server books were available, so the original author, Ron Soukup, couldn't just refer his readers to other sources for information on certain topics. Even so, some topics were not covered in the first edition, including replication and security. Ron also did not cover any details of backing up or restoring a SQL Server database, and he didn't really discuss the use and management of the transaction log.

I took over the book for SQL Server 7.0, and I completely rewrote many of the sections describing the internals of the storage engine because the entire storage engine had changed. The structure of pages, index organization, and management of locking resources were all completely different in version 7.0.

Inside Microsoft SQL Server 7.0 discussed transactions, stored procedures, and triggers all in one chapter. For the SQL Server 2000 edition, with the new feature of user-defined functions and new trigger capabilities, I split these topics into two chapters. In the 7.0 edition, query processing and tuning were covered in one huge chapter; the SQL Server 2000 edition separated these topics into two chapters, one dealing with the internals of query processing and how the SQL Server optimizer works and the other providing guidance on how to write better-performing queries. Inside Microsoft SQL Server 2000 also included many details about the workings of the transaction log, as well as an in-depth discussion of how the log is used during backup and restore operations.

Series Structure

Early in the planning stages for Inside Microsoft SQL Server 2005, I realized that it would be impossible to cover everything I wanted to cover in a single volume. My original thought was to have one volume on the storage engine components and the actual data management, and a second volume on using the T-SQL language and optimizing queries. I soon realized that this second topic itself was too big for a single volume, partly because SQL Server 2005 has so many new T-SQL features. Adequate coverage of all the new programming constructs would require a volume all its own, so at that point I invited T-SQL guru Itzik Ben-Gan to write a volume on T-SQL in SQL Server 2005. Itzik is an extremely prolific writer, and he had over 500 pages written before I completed the planning for my storage engine volume. At that point, he realized that the T-SQL language itself was too big for a single volume, and his work needed two volumes to cover everything we felt was necessary. So Inside Microsoft SQL Server 2005 is a work in four volumes.

Although one goal of ours was to minimize the amount of overlap between volumes so readers of the complete series would not have to deal with duplicate content, we also realized that not everyone would start with the same volume. Itzik and I have different approaches to describing SQL Server query processing, index use, and tuning, so when those topics are covered in more than one volume, that duplication is actually a bonus.

Inside Microsoft SQL Server 2005: T-SQL Querying

The T-SQL querying volume describes the basic constructs of the T-SQL query language and presents a thorough discussion of logical and physical query processing. It also introduces a methodology for query tuning. Itzik provides a detailed discussion of the use and behavior of all the new T-SQL query constructs, including CTEs, the PIVOT and UNPIVOT operators, and ranking functions. He covers enhancements to the TOP clause and provides examples of many new and useful ways to incorporate aggregation into your queries. New capabilities of data modification operations (INSERT, UPDATE, and DELETE) are also described in depth.

Inside Microsoft SQL Server 2005: T-SQL Programming

The T-SQL programming volume focuses on the programmability features of the T-SQL language and covers the planning and use of transactions, stored procedures, functions, and triggers in your SQL Server applications. Itzik compares set-based and cursor programming techniques and describes how to determine which technique is appropriate, and he covers CLR versus relational programming, again describing which model is appropriate for which activities. The book covers the use of temporary objects and explores the new error-handling functionality in SQL Server 2005. Itzik discusses issues with working with various datatypes, including XML data and user-defined CLR datatypes. Finally, there is a chapter on SQL Server Service Broker, which allows controlled asynchronous processing in database applications.

Inside Microsoft SQL Server 2005: The Storage Engine

The volume you are holding covers the SQL Server 2005 storage engine. I started working on this volume by taking the chapters from Inside Microsoft SQL Server 2000 that dealt with storage issues and then determining which new features were appropriate to cover. I soon realized that some reorganization was necessary, and I ended up with a full chapter on the architecture of the SQL Server 2005 engine and a whole chapter on the transaction log. As in all previous editions, I go into great depth on the actual physical storage of both data and indexes in the data files, and I describe the way that the file space is allocated and managed. Undocumented trace flags and DBCC commands are introduced where appropriate, to illustrate certain features and to allow you to confirm your understanding of SQL Server's behavior.

New features in SQL Server 2005 are pointed out as I discuss them; here are some of the most important new features covered in detail in this volume. Note that other new features are mentioned, but not all are covered in depth.

- SQL Server 2005 metadata views, including compatibility views, catalog views, and dynamic management views (and functions)
- Database snapshots
- User/schema separation
- Storage of large data objects, including row-overflow data and varchar(MAX) data
- Storage of partitioned tables and indexes
- Online index building and rebuilding
- Snapshot isolation and row-level versioning

Inside Microsoft SQL Server: Query Tuning and Optimization

This final volume in the series explains how to get the most out of the product in real applications. This was the first Inside book that I wrote as part of an authoring team, and that in itself was a very enlightening experience. Having to work with other authors was a very different experience from writing a book entirely on my own. There were good things and bad things, but the chance to learn from my coauthors definitely made the project a very positive one for me in the final analysis.

Sunil Agarwal wrote [Chapter 1](#) as basically a very extended introduction to the topic of performance, and describes a tuning methodology. The chapter covers multiple areas within a SQL Server system and application where performance problems can manifest themselves.

Adam Machanic wrote [Chapter 2](#) on SQL Server's tracing capabilities, describing the internal working of SQL Trace, as well as best practices for creating, managing, and reusing traces.

Craig Freedman wrote [Chapter 3](#), which describes the query execution process and focuses on interpreting query plans. He explains the meaning of dozens of query plan operators, in greater depth than is available anywhere else.

Craig Freedman joined me in producing [Chapter 4](#), which includes suggestions for tuning your T-SQL queries and guidelines for using the myriad of optimizer hints available in SQL Server.

I wrote [Chapter 5](#) to explain the workings of SQL Server plan caching mechanisms. I also describe how, when, and why SQL Server may choose to create new plans, and I explain how you can tell when a query is being recompiled and when an existing plan is being used. In addition, this chapter explores the new SQL Server 2005 Plan Guide feature.

Ron Talmage wrote [Chapter 6](#), and provides details on troubleshooting concurrency problems. He addresses both optimistic and pessimistic concurrency, and discusses the methods of problem detection and suggestions for alleviating the problems, including dozens of scripts using SQL Server's metadata. Finally, he provides some best-practice guidelines for making the best choice for concurrency modeling your applications.

Examples and Scripts

Many of the features and behaviors described in this volume are illustrated using T-SQL code. Some of the code is just a few lines long, but other examples require very complex coding, including multiway joins of some of the dynamic management views, all of which have impossibly long and hard-to-type names.

All code samples longer than a couple of lines are available for download from the companion Web site at <http://www.InsideSQLServer.com/companion>.

Topics Not Covered

As I mentioned in the introduction to the Storage Engine volume, even in four volumes certain features and aspects of the product cannot be covered. Also keep in mind that the books in the series are not intended to be how-to books for database administrators or for database application programmers. They are intended to explain how SQL Server works behind the scenes, so you will have a solid foundation on which to build and troubleshoot your applications and will understand why SQL Server behaves the way it does.

In addition to business intelligence (Analysis Services, Integration Services, Reporting Services) and high availability (replication, database mirroring, log shipping, clustering), other topics that are beyond the scope of this book include:

- Notification Services
- Security
- XML indexes
- Full-text search
- Client programming interfaces

Caveats and Disclaimers

To illustrate some of the behaviors of SQL Server, this volume discusses some undocumented features of the product or undocumented objects, such as internal tables. Some of these are potentially "discoverable" on your own, usually by looking at the definition of the supported functions, procedures, or views. In those cases, I am simply saving you time by providing information that you could have eventually discovered on your own. Another category of undocumented features is undocumented DBCC commands or trace flags, which I introduce only for the purpose of allowing deeper analysis or more thorough observation of certain product behavior. For the most part, these are not discoverable unless someone tells you about them. Please keep in mind that undocumented means unsupported. This means that if you have additional questions about an undocumented feature that I describe, you cannot call Customer Support Services at Microsoft and expect the representative on the phone to answer them. There is also no guarantee that an undocumented feature will continue to behave in the same way in the next version of SQL Server. In some cases, undocumented features can change behavior in a service pack, and Microsoft will not be obligated to tell you about this change in a readme file or a Knowledge Base article. Throughout this volume, I will let you know when I refer to features or tools that are undocumented, and in some cases I will also reiterate that Microsoft provides no support for them. However, consider this a global caveat for all such undocumented features.

How to Get Support

Every effort has been made to ensure the accuracy of this book's content. If you run into problems, you can refer to one of the sources listed below.

Companion Web Site

Despite my best intentions, as well as a review by a couple of awesome technical reviewers and members of the SQL Server team at Microsoft, this book is not perfect, as no book is. Updates and corrections will be posted on the companion Web site at <http://www.InsideSQLServer.com/companion>. In addition, if you find anything you think is incorrect, feel free to use the feedback form on that site to inform me of the problem.

Microsoft Learning

Microsoft also provides correction for books at the following Web address:
<http://www.microsoft.com/learning/support>.

To connect directly with the Microsoft Learning Knowledge Base and enter a query regarding an issue you have encountered, you can go to <http://www.microsoft.com/learning/support/search.asp>.

In addition to sending feedback to the author, you can send comments or questions to Microsoft using either of the following methods:

Postal Mail:

Microsoft Learning
 Attn: Inside Microsoft SQL Server 2005 Editor
 One Microsoft Way
 Redmond, WA 98052-6399

E-mail:

mspinput@microsoft.com

Please note that product support is not offered through the above addresses. For SQL Server support, go to <http://www.microsoft.com/sql>. You can also call Standard Support at 425-635-7011 weekdays between 6 A.M. and 6 P.M. Pacific time, or you can search Microsoft's Support Online at <http://support.microsoft.com/default.aspx>.

I hope you find value in this book even if I haven't covered every single SQL Server topic in which you are interested. You can let me know what you'd like to learn more about, and I can perhaps refer you to other books or whitepapers. Or maybe I'll write about it on my blog at http://sqlblog.com/blogs/kalen_delaney or write an article for SQL Server Magazine. You can contact me via my Web site at <http://www.InsideSQLServer.com>.

Chapter 1. A Performance Troubleshooting Methodology

â Sunil Agarwal

In this chapter:	
Factors That Impact Performance	2
Troubleshooting Overview	14
Summary	55

SQL Server is an enterprise-level database server that has been used by organizations to run their mission-critical applications worldwide. These applications impose the toughest requirements in terms of availability, performance, and scalability. Very few enterprise workloads, if any, have requirements that exceed these. Microsoft SQL Server 2005 has successfully met or exceeded the requirements of these workloads under demanding conditions. One such deployment of SQL Server is NASDAQ, which may be familiar to most readers. It runs SQL Server 2005 on two, 4-node Dell PowerEdge 6850 clusters to support its Market Data Dissemination System (MDDS). Every trade that is processed in NASDAQ marketplace goes through MDDS, with SQL Server 2005 handling 5,000 transactions per second at market open. So if you are experiencing performance/scalability issues with your application(s), chances are that it is not the SQL Server but something else. The trick is to identify what it is (that is, to make a diagnosis) and how to fix it (that is, to troubleshoot) so that your application runs smoothly again. Diagnosing and troubleshooting performance problems cannot be done on an ad hoc basis. You need a strategy/methodology to diagnose and troubleshoot the performance problem(s) quickly to minimize the time your applications are unavailable or any slowdown is experienced by your users.

You may wonder what we mean by performance because the word may mean different things to different people. For the discussions here, we will focus on the following three terms.

- Response time Refers to the interval between the time when a request is submitted and when the first character of the response is received.

- Throughput Refers to the number of transactions that can be processed in a fixed unit of time.
- Scalability Refers to how the throughput and/or the response time changes as we add more hardware resources. In simple terms, scalability means that if you are hitting a hardware bottleneck, you can alleviate it simply by adding more resources.

Factors That Impact Performance

Most users perceive the performance of their SQL Server based on the responsiveness of their application. While SQL Server itself does play a part in application responsiveness, before we can isolate SQL Server as the source of a performance issue, we first need to understand the factors that impact the performance of your application. We will look into these factors for completeness before we switch our focus to troubleshooting performance problems in SQL Server.

At a high level, many factors can impact the performance and scalability that can be achieved in your application. We will be looking at the following:

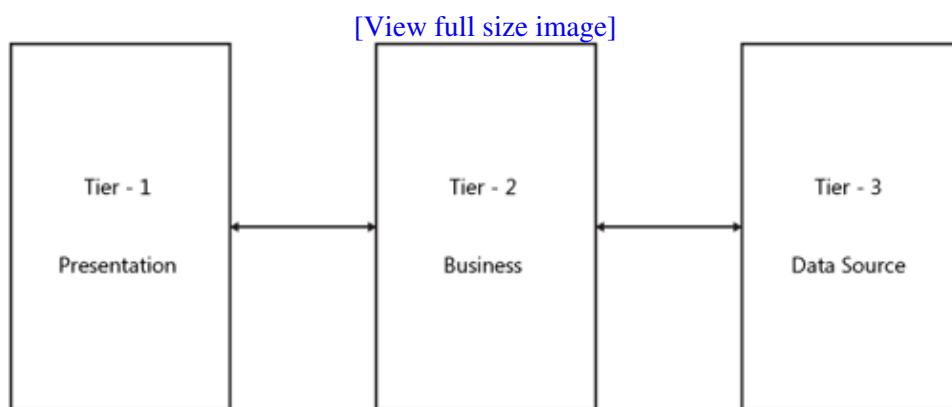
- Application Architecture
- Application Design
- Transactions and Isolation Levels
- Transact-SQL Code
- Hardware Resources
- SQL Server Configuration

Application Architecture

Most applications today use multitier architecture for better availability, performance, and scalability. Multitier architecture offers a huge improvement, as you will see later in this section, over the two-tier (in other words, client-server) model that was prevalent in the 1980s and early 1990s. These tiers define the logical boundary of the application, and they can run on the same physical box or multiple boxes, though the most common deployment is to run these tiers on their own box(es). For an application to perform well, all three tiers need to function without any bottlenecks; otherwise users may experience a slowdown in the performance and response time.

[Figure 1-1](#) shows a typical application with three tiers.

Figure 1-1. Typical application with three tiers.



The first tier, also known as the client tier, is the presentation layer that provides the users an interface to interact with the application. Common ways to interact with the application are using a Web browser, a custom application program interface (API), or through a command line. Depending on the application, you can have anywhere from a small number of users to very large numbers of users interacting with the application concurrently. For example, if you consider the MSN home page, millions of users can interact with the application at the same time. Typically, users can experience performance issues in this tier for the following reasons:

- The network bandwidth is not sufficient for content delivery. We all likely have experienced occasional slowdown when downloading contents. This can potentially be caused by the network bandwidth at the client or performance bottleneck at tier-2.
- The client machine is not powerful enough to deliver the content efficiently.

We will not discuss performance issues in tier-1 in any detail as this is outside the scope of this book.

The second tier, also known as the middle/application tier, hosts the business logic of the application. A typical middle tier has two main components: a Web server to create HTTP-based contents and an application server for hosting the business logic. Depending on the type of application, the business logic can range from simple to very complex. It is not uncommon for a complex application to have a few million lines of code. This tier interacts with the storage layer, the tier-3, for storing and accessing persisted data. For example, an order management application stores the orders in the database. When a user wants to access information about any specific order, the middle tier retrieves this information by querying the database. Besides business logic, the middle tier can be leveraged to provide better performance and scalability to the application. Some common ways are the following:

- By caching the results of common client requests to minimize recomputations. This would not be possible with two-tier architecture.
- By using multiple servers to host this tier so as to distribute the workload for better scalability and availability. This would not be possible with two-tier architecture.
- By pooling connections (that is, sessions) to a SQL Server and reusing them to execute database requests on behalf of application users. This is possible because not all active users need to access the database at the same time. Connection pooling offers performance improvement at two levels. First, it minimizes that number of connections/sessions that the database server, in our case a SQL Server, needs to manage. Each connection takes approximately 50 KB of memory. As you can imagine, if the application tier opens a separate database connection for each of its million active users, it can be a huge drain on the SQL Server memory. Second, it takes both time and CPU resources to open and close a session every time to access SQL Server. Using connection pooling, you can minimize the number of required SQL Server connections and also minimize the memory overhead of connections. You will need to analyze your application workload to find the appropriate number of connections for the pool. Connection pooling is not possible with two-tier architecture as each user of the application has to establish its own connection, thereby limiting the number of concurrent users that can be supported by an application.

The third tier is the data tier. This tier is responsible for storing, retrieving, and manipulating the data needed by the application. Though this tier does not have to be a database server, more commonly it is. We will refer to this tier as the database tier from now on. We will also use this term synonymously with SQL Server as we are focusing on performance of applications that are running on SQL Server. Like other tiers, the database tier is critical to your application's performance. If the application tier stalls while accessing data from the database tier, users will experience slowdown. We will cover more details on the database tier later in this chapter.

Application Design

Like any other software, the design of your application is critical to its performance. There is no yardstick to classify an application as good or bad, but typically if the application is stalling even though the hardware resources are not fully utilized, you have a problem. For sake of brevity, let us divide application logic into two parts. The first part contains the interaction of the application with the database tier. The second part contains everything else in the application that runs on the middle tier such as the business logic, workflow, security, session management, caching, and so on. As you can imagine, both parts are critical to the performance and scalability of the application, but we will focus on the first part as we are looking into performance issues that relate to the database tier (in our case, the SQL Server). We'll now look at some of the common application design considerations for the first part that may impact the performance of your application.

Database Schema and Its Physical Design

A poorly designed database schema can lead to inefficient queries and data modification operations. We need to be concerned about three key factors.

The first factor is the normal form of your database schema. Database theory primarily defines five normal forms for your database schema. The higher the normal form, the easier it is to maintain the database consistency. The main goal of database normal forms is to eliminate the data redundancy. By eliminating repeating or multiple copies of the same data, you can achieve better performance of data modification operations because you will need to modify it in only one place, which leads to fewer locks and logging. Though it is desirable to have fully normalized database schema, it can add complexity to your application design. A higher normal form implies more tables; hence, your application will need to more join operations to retrieve the same information. A query involving more join operations is relatively complex and can take more system resources to optimize and execute, which ultimately slows down the performance of the query and of the other concurrent operations in the database system.

To illustrate how schema normalization can impact the cost of data modification operations, consider a simple schema as shown in the following example. It shows two schemas that store the same information about students in the database. In the first schema, only one table represents Student, but in schema-2, it has been broken into two tablesâ€”Student and Dept.

```

Schema-1:
Student (
    student_id          int,
    student_name        varchar(100),
    student_status      int,
    dept_name           varchar(100)
    dept_bldg_no       int,
    dept_phone          char(10))

Schema-2:
Student (
    student_id          int,
    student_name        varchar(100),
    student_status      int,
    dept_id              int)

Dept (
    dept_id              int,
    dept_name           varchar(100),
    dept_bldg_no       int,
    dept_phone          char(10))

```

Let us consider the performance of the following two operations under each of the two schemas:

- To project (for example, Select) student and department information In schema-1, you can get this information by simply scanning the table Student without requiring any joins. In schema-2, you will need to join the Student and Dept. tables.
- To update the telephone number for a department In schema-1, you will potentially need to update multiple rows in the Student table which requires more locks and log records. In schema-2, you will need to update only one row in the Dept table.

Because of data redundancy, the size of the database for schema-1 will be larger than schema-2, which can also impact the performance of your application. So which schema should be chosen by the application? The answer actually lies with the type of operations done by the application. For example, if the application is predominantly reading data (for example, a data warehouse application), then schema-1 may be okay.

The second factor is the availability of the useful indexes. A useful index is defined to be an index that can be chosen by the optimizer to process a Data Manipulation Language (DML) statement in your application. Absence of a useful index may cause a SQL Server to scan the full table, which can negatively impact the performance of the statement. For example, if there is an index on column student_name in the schema-1 above, SQL Server can locate the student by his or her name without resorting to a full-table scan. On the other hand, if the index is not useful but it exists, it can add unnecessary overhead for data modification operations, as the index entries need to be maintained to account for any changes in the table. You can take the corrective action by identifying indexes that are useful, adding the ones that are missing, and by dropping the ones that are not useful.

The third factor is the mapping of tables and databases to the physical disks or Logical Unit Numbers (LUNs), as in a Storage Area Network (SAN) environment. For the discussion here, we will use the term physical disk, but it is similar to LUNs in a SAN environment. You need to understand the access pattern of the data and then distribute the databases and tables among physical disks so as to maximize the use of available I/O bandwidth. Some examples of nonoptimal mappings are explored here.

Mixing log and data onto the same physical disk

Most operations to log files are sequential, whereas the access to data is inherently random, depending upon user queries and data modification operations. Log records are written sequentially and are accessed sequentially when they are backed up. The only other time you need to access log records in reverse order is when rolling back a transaction, which is not a common operation. Note that in SQL Server 2005, triggers are implemented using row versioning. In previous versions of SQL Server, triggers were implemented by scanning the log records in reverse order, which can cause the disk head to move back and forth, thereby impacting the I/O throughput of the physical disk containing log files. A general recommendation is to put log files onto their own physical disk. If this is not possible, you should focus on ensuring that healthy latency is maintained on log writes, as this directly impacts the response time.

Sharing physical disk(s) among and user databases

The tempdb is a shared database, one per instance of SQL Server, that is used both by the application and SQL Server to store/manipulate temporary data. You need to make sure that the tempdb is created on the disk I/O subsystem that supports the I/O bandwidth needed for the workload. If you create tempdb on a slower disk, it can severely impact the performance of the application. A general recommendation is to create tempdb on its own physical disk(s).

Mapping heavily accessed tables onto the same physical disk

A general recommendation here is to distribute I/O load across multiple physical disk(s). So in this case, you may want to consider mapping these tables to different physical disks to reduce the likelihood of an I/O bottleneck.

Transactions and Isolation Levels

Applications interact with SQL Server using one or more transactions. A transaction can be started explicitly by executing the BEGIN TRANSACTION Transact-SQL statement or implicitly, by SQL Server, for each statement that is not explicitly encapsulated by the transaction. Transactions are very fundamental to database systems. A transaction represents a unit of work that provides the following four fundamental properties:

- Atomicity Changes done under a transaction that are either all commit or are rolled back. No partial changes are allowed. It is an all-or-nothing proposition.
- Consistency Changes done under a transaction database from one consistent state to another. A transaction takes a database from one consistent state to another.
- Isolation Changes done by a transaction are isolated from other concurrent transactions until the transaction commits.
- Durability Changes done by committed transactions are permanent.

SQL server uses locks to implement transaction isolation. It acquires an exclusive (X) lock on the data before modifying it. An exclusive lock guarantees that two transactions cannot modify the data at the same time. The exclusive lock is held for the duration of the transaction so that concurrent transactions can read or modify the data only after the first transaction commits or rolls back. Similarly, a transaction acquires a shared (S) lock before reading the data. As the name share lock implies, it allows multiple transactions to share a resource concurrently. In other words, concurrent transactions can read the same data by acquiring a shared lock of their own without blocking each other. [Table 1-1](#) shows a simple lock compatibility table. It shows that two shared locks don't conflict with each other, but an exclusive lock conflicts both with shared and exclusive locks.

Table 1-1. A Simple Lock Compatibility Table

Lock Mode	shared (S)	exclusive (X)	shared (S)	OK	NO	exclusive (X)	NO	NO
shared (S)	X	X	X	X	X	X	X	X
exclusive (X)	X	X	X	X	X	X	X	X
shared (S)	X	X	X	X	X	X	X	X
exclusive (X)	X	X	X	X	X	X	X	X
NO	X	X	X	X	X	X	X	X
NO	X	X	X	X	X	X	X	X
exclusive (X)	X	X	X	X	X	X	X	X
NO	X	X	X	X	X	X	X	X

As you can imagine, blocking among transactions can impact the performance of your application significantly, but it is necessary to guarantee transaction isolation property as described earlier. For this reason, the SQL Server provides

- Many more locking modes (for example, intent locks) at various granularities (for example, row, page, table) to maximize concurrency.
- Locking hints so that applications can control the locking behavior at an object level.
- Optimized code path so that acquiring/releasing locks is efficient.

As an application designer, you need to pay particular attention to minimize blocking. One interesting aspect of the isolation guarantee is the level of isolations. The SQL-99 standard defines four isolation levels that can be used to run a transaction. The transaction isolation levels provide applications with an option to choose between consistency and concurrency. The higher the consistency, the lower the concurrency. A brief description of these isolation levels follows. [Chapter 6](#), "Concurrency Problems," covers isolation levels in more detail.

- **READ UNCOMMITTED** A transaction executing under this isolation level does not need to acquire locks to read user data. This means that a transaction can read the data that is potentially modified by

an active concurrent transaction without getting blocked. Since we won't know the state or the fate of the transaction that has modified the data, the application design using this isolation level must be resilient to this fact.

- **READ COMMITTED** This is the default isolation level in SQL Server. A transaction executing under this isolation level can only read the committed data and will get blocked if a concurrent transaction is modifying the data. This is accomplished by requesting the S lock on the data before reading it. However, SQL Server releases the S lock after the data has been read and does not hold it for the duration of the transaction. This allows a concurrent transaction to modify the data, even though the first transaction did not commit yet. Since the S lock is not held for the duration of the transaction, if the same data is read again, there is no guarantee that it will be the same. For example, it may have been changed by other concurrent transactions that started after the first read and committed before the second read. In other words, there is no guarantee that the read is repeatable.
- **REPEATABLE READ** This isolation level guarantees that the data read by the transaction will not change for its duration. SQL Server accomplishes this by holding the S lock for the duration of the transaction. As you can see, this isolation level offers a higher consistency to your application, but it does so at the cost of concurrency (in other words, more blocking). Now, if a concurrent transaction wants to modify the data, it has to wait for the first transaction to complete. While this isolation level guarantees repeatable read, it does not prevent phantoms. For example, if a new row gets inserted by a concurrent transaction and that qualifies the search criteria of the first transaction, this new row will show up in the result set when queried again once the transaction that inserted the new row commits.
- **SERIALIZABLE** This is the highest level of consistency offered by isolation levels. A transaction running at this isolation level reads only the committed data that does not change for the duration of the transaction, and there are no phantoms. For the purist, a serializable transaction represents a serial sequence of execution of concurrent transactions. SQL Server implements it either using key-range locks or by acquiring locks at a higher granularity, for example at table level. A transaction running at this isolation level causes the most blocking.

One thing to keep in mind is that a transaction acquires an X lock when modifying data, and this lock is held for the duration of the transaction, regardless of the isolation level of the transaction.

When designing an application you need to be cognoscente of the amount of time that the transactions will stay open. The duration of the transaction determines the duration of X locks, as these locks are held until the transaction is finished. For S locks, the transaction isolation level determines the duration of the lock and its granularity. If your application is running transactions at an isolation level higher than the default READ COMMITTED, your S locks will be held until the transaction is finished. The longer a lock is held, the higher the chances that it will block concurrent transactions, thereby impacting the throughput and the response time of your application.

At first glance, it may seem that most applications should use an isolation level that offers the most consistency. But in practice, the commonly used isolation level is READ COMMITTED and, in fact, this is the default isolation provided by SQL Server. A higher isolation level should only be chosen after careful consideration.

Let us consider a simple hotel reservation application and explore couple of ways to implement it using different transaction isolation levels.

Assume your room reservation application is a Web-based application that displays room information, including availability and the nightly rate. This application allows online customers to reserve a room and also allows hotel staff to change the nightly rate of the room depending upon the season and current vacancy rate. Let us assume that there is one row per room per day of the year. One of the main design considerations is to prevent two customers from reserving the same room. We'll look at two possible ways to implement a solution.

First Implementation

A user searches for a room meeting his/her criteria. The application queries the database and displays the rooms that satisfy the criteria. To guarantee that the information about the room(s) does not change, the transaction is run at a repeatable read isolation level that obtains S lock(s) on the row(s) representing the room(s). The user decides to reserve one room, provides the payment information, and then submits the Web-form. Because an S lock was held on the row representing the room, no one else would have reserved this room, the application updates the reservation status of the room, and then commits the transaction. The customer is informed that the reservation was successful.

There are a few issues with this implementation. First, the transaction is active while waiting for the customer to browse and update the room information. This means that the transaction duration is unlimited, as it depends upon when the customer submits the form. A long-running transaction means that the lock acquired will be held for the longer duration, as well leading to more blocking if there are concurrent transactions that need to modify this data. Second, if two customers browsing the same room want to reserve it, a deadlock will occur because the transactions representing the customers will hold the S lock on the row, and now both want to acquire X lock on it. The customers will end up waiting for each other leading to a deadlock. You can eliminate deadlock by doing the SELECT with UPDLOCK hint, but you will still have blocking.

Second Implementation

A user interacts with the application as in the previous implementation, but the application is in the READ COMMITTED isolation level and returns the room meeting the search criteria without holding any locks and completes the transaction. However, to prevent concurrent reservations by multiple customers, a new column timestamp is added to the schema that tracks the time of the last update. So when a customer submits the room reservation request, the application verifies if the timestamp of the row to be updated is still the same. If it is, the room reservation is successful; otherwise, it fails and the customer is given the message "Sorry, the room is already taken." The interesting part about this implementation is that it avoids lost-update (that is, multiple customers reserving the same room) while running the transactions for a shorter duration at a lower isolation level. So it is a preferred design over the first implementation.

This example illustrates one of many poor application designs that can lead to unnecessary blocking. If you see a lot of blocking in your application, you will need to pay close attention to the duration of transactions and the isolation level used. In SQL Server 2005, you may also consider using SNAPSHOT ISOLATION or READ_COMMITTED_SNAPSHOT to remove blocking between readers and writers (for example, SELECT and UPDATE). [Chapter 6](#) covers isolation levels in more detail.

Transact-SQL Code

Though the SQL Server optimizer does rearrange Transact-SQL submitted for better optimization, it cannot compensate for a poor design. One such example is using the cursor to update multiple rows. Because the cursor operates on one row at time, it will be significantly slower than the corresponding set operation. Here is one example to illustrate this point.

Code View:

```

CREATE TABLE t1 (c1 int PRIMARY KEY, c2 int, c3 char(8000))
GO
â   Load 6000 rows into this table as follows
DECLARE @i int
SELECT @i = 0
WHILE (@i < 6000)
BEGIN
    INSERT INTO t1 VALUES (@i, @i + 1000, 'hello')
    SET @i = @i + 1
END

```

â Method-1: Update all the rows in this table in a single statement

```
BEGIN TRAN
UPDATE t1 SET c2 = 1000 + c2
COMMIT TRAN
```

â Method-2: Update all the rows in this table using cursor

```
DECLARE mycursor CURSOR FOR
    SELECT c2 FROM t1
OPEN mycursor
GO
BEGIN TRAN
FETCH mycursor
WHILE (@@FETCH_STATUS = 0)
BEGIN
    UPDATE t1 SET c2 = 1000 + c2 WHERE CURRENT OF mycursor
    FETCH mycursor
END
COMMIT TRAN
â Now query the total worker time
SELECT TOP 10
    total_worker_time/execution_count AS avg_cpu_cost,
    execution_count,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(max), text)) * 2
            ELSE statement_end_offset
        END - statement_start_offset)/2)
    FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM sys.dm_exec_query_stats
ORDER BY [avg_cpu_cost] DESC
```

â here is the output of the DMV query

avg_cpu_cost	exec_count	query_text
4913844	1	UPDATE [t1] SET [c2] = @1+[c2]
58634	1	SELECT top 10 total_worker_tim
8437	6000	FETCH mycursor
5641	1	FETCH mycursor
5268	6000	UPDATE t1 SET c2 = 1000 + c2 w

In this case, the updates using the cursor are taking $(8,437 * 6,000 + 5,268 * 6,000) = 81,690,000$ units. This is a factor of approximately 20, compared to 4,913,844 units taken by set-oriented update. It is unlikely that an application will use a cursor to update a large set of rows when it can alternatively use a set-oriented Transact-SQL UPDATE statement as described previously. However, the point is that poorly designed Transact-SQL can lead to performance issues in your application, and it is worth examining.

Hardware Resources

Hardware resources are the horsepower you need to run your application. It will do you no good if you have a well-designed application, but it is running on hardware that is not on a par with the demands of the workload. Most often, an application is tested in a small-scale environment with simulated workload. While this does serve a useful purpose, clearly it is not a replacement for measuring the performance of the workload and the hardware it actually runs on in production. A study by the Morse Consultancy Group

(<http://www.morse.co.uk>) indicates that 89 percent of the organizations have inadequate processes and hardware for performance testing. So when your application hits performance issues, it can be the hardware, however, that is not necessarily so. Any hardware resource (CPU, I/O, memory, or network) that is pushed beyond its operational capacity in any application tier will lead to a slowdown in your application. Note, too, that hardware bottlenecks may also be caused by poor application designs, which ultimately need to be addressed. In such cases, upgrading hardware is a short-term solution at best.

SQL Server Configuration

SQL Server is designed to be self-tuning to meet the challenges of the workload. In most cases, it just works well with out-of-the-box configuration settings. However, in some cases, you may need to tweak configuration parameters for maximum performance. Inside SQL Server 2005: The Storage Engine (Microsoft Press, 2006) has some additional details about configuration options. In this section, we will list the options that you will most likely need to monitor to track down performance issues. The relevant configuration options can be considered to be either CPU-related or memory-related options.

CPU-Related Configuration Options

The CPU-related configuration options are used, for example, to control the number of CPUs or sockets that can be used by a SQL Server instance, maximum degree of parallelism, and the number of workers. Some commonly used options in this category include the following:

- **Affinity Mask** This option can be used to control the mapping of CPUs to the SQL Server process. By default, a SQL Server uses all processors available on the Server box. A general recommendation is not to use affinity mask option, as SQL Server performs best in the default setting. You may, however, want to use this option under two situations.
 - ◆ First, if you are running other applications on the box, the Windows operating system may move around process threads to different CPUs under heavy load. By using affinity masks, you can bind each SQL Server scheduler to its own CPU. This can improve performance by eliminating thread migration across processors, thereby reducing context switching.
 - ◆ Second, you can use this parameter to limit the number of CPUs on which a SQL Server can run. This is useful if you are running multiple SQL Server instances on the same Server box and want to limit CPU resources taken by each SQL Server and to minimize their interference.
- **Lightweight Pooling** When this configuration is enabled, SQL Server makes use of Windows fibers. A worker can map to a Windows thread or to a fiber. A fiber is like a thread, but it is cheaper than normal thread because switching between two workers (that is, fiber threads) can be done in user mode instead of kernel mode. So if your workload is experiencing a CPU bottleneck with significant time spent in kernel mode and in context switching, you may benefit by enabling this option. You must test your workload with this option before enabling it in the production system, as more often than not this option may cause performance regression. You should also keep in mind that CLR integration is not supported under lightweight pooling.
- **Max Worker Threads** You can think of workers as the SQL Server threads that execute user or batch requests. A worker is bound to a batch until it completes. So the maximum number of workers limits the number of batches that can be executed concurrently. By default, SQL Server sets the Max Worker Threads as described in the following table.

Number of CPUs	32-bit computer	64-bit computer
<= 4 processors	256	512
8 processors	288	576

16 processors	352	704
32 processors	480	960

For most installations, this default setting is fine. Each worker takes 512-KB memory on a 32-bit, 2 MB on X64, and 4 MB on IA64. To preserve memory, the SQL Server starts off with a smaller number of workers. The pool of workers grows or shrinks based on the demand. You may want to change this configuration under two conditions. The first is if you know that your application uses a smaller number of workers. By configuring it to a lower number, the SQL Server does not need to reserve memory for the maximum number of workers. The second is if you have many long-running batches (presumably involving lock waits), such that the number of workers needed may exceed the default configuration. This is not a common case, and you may want to look at your application design to analyze this.

- **Max Degree of Parallelism** This configuration parameter controls the maximum number of processors or cores that can be deployed to execute a query in parallel. Parallel queries provide better response time but take more CPU resources. You need to look into this configuration parameter if you are encountering CPU bottleneck, described later in this chapter.

Memory-Related Configuration Options

The memory-related configuration options are used to control the memory consumed by SQL Server. Some of the commonly used configuration options in this category include the following:

- **Max and Min Server Memory** This is perhaps the most critical of the configuration options, especially in 32-bit configurations, from a performance perspective. This configuration parameter is often confused with the total memory configured for a SQL Server, but it is not the same. It represents the configured memory for the buffer pool. SQL Server, or any database server for that matter, is a memory-hungry application. You want to make as much memory available for SQL Server as possible. The recommendation on 64 bits is to put an upper limit in place to reserve memory for the OS and allocations that come from outside the Buffer Pool. You will also need to cap the memory usage by SQL Server when other applications (including other instances of SQL Server) are running on the same Server box.
- **AWE Enabled** On a 32-bit box, the SQL Server process can only address 2 GB of virtual memory, or 3 GB if you have added the /3 GB parameter to the boot.ini file and rebooted the computer, allowing the /3 GB parameter to take effect. If you have physical memory greater than 4 GB and you have enabled this configuration option, the SQL Server process can make use of memory up to 64 GB normally, and up to 16 GB if you have used /3 GB parameter. Additionally, the SQL Server process requires Lock Pages in Memory privilege in conjunction with AWE-Enabled option. There are some restrictions in terms of the SQL Server SKU and the version of the Windows operating system under which you are running. Please refer to SQL Server 2005 Books Online (BOL) on TechNet for more details. You should be aware of certain key things when using the AWE option. First, though it allows the SQL Server process to access up to 64 GB of memory for the buffer pool; the memory available to query plans, connections, locks, and other critical structures is still limited to less than 2 GB. Second, the AWE-mapped memory is nonpageable and can cause memory starvation to other applications running on the same Server box. Starting with SQL Server 2005, the AWE memory can be released dynamically, but it still cannot be paged out. SQL Server may release this memory in response to physical memory pressure. Third, this option is not available on a 64-bit environment. However, if the SQL Server process has been granted Lock Pages in Memory privilege, the buffer pool will lock pages in memory and these pages cannot be paged out.

So far, we have discussed some of the major considerations when designing and deploying a database application, particularly in the context of SQL Server, though most of the discussion is also applicable to other database servers as well.

Chapter 1. A Performance Troubleshooting Methodology

â Sunil Agarwal

In this chapter:	
Factors That Impact Performance	2
Troubleshooting Overview	14
Summary	55

SQL Server is an enterprise-level database server that has been used by organizations to run their mission-critical applications worldwide. These applications impose the toughest requirements in terms of availability, performance, and scalability. Very few enterprise workloads, if any, have requirements that exceed these. Microsoft SQL Server 2005 has successfully met or exceeded the requirements of these workloads under demanding conditions. One such deployment of SQL Server is NASDAQ, which may be familiar to most readers. It runs SQL Server 2005 on two, 4-node Dell PowerEdge 6850 clusters to support its Market Data Dissemination System (MDDS). Every trade that is processed in NASDAQ marketplace goes through MDDS, with SQL Server 2005 handling 5,000 transactions per second at market open. So if you are experiencing performance/scalability issues with your application(s), chances are that it is not the SQL Server but something else. The trick is to identify what it is (that is, to make a diagnosis) and how to fix it (that is, to troubleshoot) so that your application runs smoothly again. Diagnosing and troubleshooting performance problems cannot be done on an ad hoc basis. You need a strategy/methodology to diagnose and troubleshoot the performance problem(s) quickly to minimize the time your applications are unavailable or any slowdown is experienced by your users.

You may wonder what we mean by performance because the word may mean different things to different people. For the discussions here, we will focus on the following three terms.

- Response time Refers to the interval between the time when a request is submitted and when the first character of the response is received.
- Throughput Refers to the number of transactions that can be processed in a fixed unit of time.
- Scalability Refers to how the throughput and/or the response time changes as we add more hardware resources. In simple terms, scalability means that if you are hitting a hardware bottleneck, you can alleviate it simply by adding more resources.

Factors That Impact Performance

Most users perceive the performance of their SQL Server based on the responsiveness of their application. While SQL Server itself does play a part in application responsiveness, before we can isolate SQL Server as the source of a performance issue, we first need to understand the factors that impact the performance of your application. We will look into these factors for completeness before we switch our focus to troubleshooting performance problems in SQL Server.

At a high level, many factors can impact the performance and scalability that can be achieved in your application. We will be looking at the following:

- Application Architecture
- Application Design
- Transactions and Isolation Levels
- Transact-SQL Code

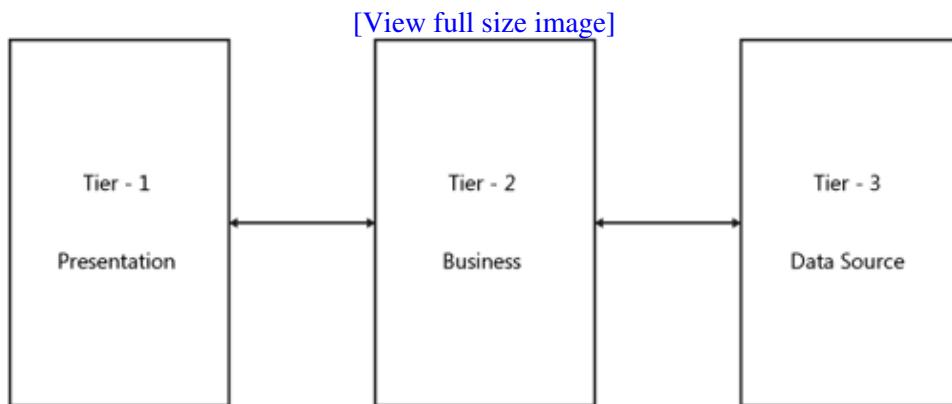
- Hardware Resources
- SQL Server Configuration

Application Architecture

Most applications today use multitier architecture for better availability, performance, and scalability. Multitier architecture offers a huge improvement, as you will see later in this section, over the two-tier (in other words, client-server) model that was prevalent in the 1980s and early 1990s. These tiers define the logical boundary of the application, and they can run on the same physical box or multiple boxes, though the most common deployment is to run these tiers on their own box(es). For an application to perform well, all three tiers need to function without any bottlenecks; otherwise users may experience a slowdown in the performance and response time.

[Figure 1-1](#) shows a typical application with three tiers.

Figure 1-1. Typical application with three tiers.



The first tier, also known as the client tier, is the presentation layer that provides the users an interface to interact with the application. Common ways to interact with the application are using a Web browser, a custom application program interface (API), or through a command line. Depending on the application, you can have anywhere from a small number of users to very large numbers of users interacting with the application concurrently. For example, if you consider the MSN home page, millions of users can interact with the application at the same time. Typically, users can experience performance issues in this tier for the following reasons:

- The network bandwidth is not sufficient for content delivery. We all likely have experienced occasional slowdown when downloading contents. This can potentially be caused by the network bandwidth at the client or performance bottleneck at tier-2.
- The client machine is not powerful enough to deliver the content efficiently.

We will not discuss performance issues in tier-1 in any detail as this is outside the scope of this book.

The second tier, also known as the middle/application tier, hosts the business logic of the application. A typical middle tier has two main components: a Web server to create HTTP-based contents and an application server for hosting the business logic. Depending on the type of application, the business logic can range from simple to very complex. It is not uncommon for a complex application to have a few million lines of code. This tier interacts with the storage layer, the tier-3, for storing and accessing persisted data. For example, an order management application stores the orders in the database. When a user wants to access information

about any specific order, the middle tier retrieves this information by querying the database. Besides business logic, the middle tier can be leveraged to provide better performance and scalability to the application. Some common ways are the following:

- By caching the results of common client requests to minimize recomputations. This would not be possible with two-tier architecture.
- By using multiple servers to host this tier so as to distribute the workload for better scalability and availability. This would not be possible with two-tier architecture.
- By pooling connections (that is, sessions) to a SQL Server and reusing them to execute database requests on behalf of application users. This is possible because not all active users need to access the database at the same time. Connection pooling offers performance improvement at two levels. First, it minimizes that number of connections/sessions that the database server, in our case a SQL Server, needs to manage. Each connection takes approximately 50 KB of memory. As you can imagine, if the application tier opens a separate database connection for each of its million active users, it can be a huge drain on the SQL Server memory. Second, it takes both time and CPU resources to open and close a session every time to access SQL Server. Using connection pooling, you can minimize the number of required SQL Server connections and also minimize the memory overhead of connections. You will need to analyze your application workload to find the appropriate number of connections for the pool. Connection pooling is not possible with two-tier architecture as each user of the application has to establish its own connection, thereby limiting the number of concurrent users that can be supported by an application.

The third tier is the data tier. This tier is responsible for storing, retrieving, and manipulating the data needed by the application. Though this tier does not have to be a database server, more commonly it is. We will refer to this tier as the database tier from now on. We will also use this term synonymously with SQL Server as we are focusing on performance of applications that are running on SQL Server. Like other tiers, the database tier is critical to your application's performance. If the application tier stalls while accessing data from the database tier, users will experience slowdown. We will cover more details on the database tier later in this chapter.

Application Design

Like any other software, the design of your application is critical to its performance. There is no yardstick to classify an application as good or bad, but typically if the application is stalling even though the hardware resources are not fully utilized, you have a problem. For sake of brevity, let us divide application logic into two parts. The first part contains the interaction of the application with the database tier. The second part contains everything else in the application that runs on the middle tier such as the business logic, workflow, security, session management, caching, and so on. As you can imagine, both parts are critical to the performance and scalability of the application, but we will focus on the first part as we are looking into performance issues that relate to the database tier (in our case, the SQL Server). We'll now look at some of the common application design considerations for the first part that may impact the performance of your application.

Database Schema and Its Physical Design

A poorly designed database schema can lead to inefficient queries and data modification operations. We need to be concerned about three key factors.

The first factor is the normal form of your database schema. Database theory primarily defines five normal forms for your database schema. The higher the normal form, the easier it is to maintain the database consistency. The main goal of database normal forms is to eliminate the data redundancy. By eliminating repeating or multiple copies of the same data, you can achieve better performance of data modification operations because you will need to modify it in only one place, which leads to fewer locks and logging.

Though it is desirable to have fully normalized database schema, it can add complexity to your application design. A higher normal form implies more tables; hence, your application will need to perform more join operations to retrieve the same information. A query involving more join operations is relatively complex and can take more system resources to optimize and execute, which ultimately slows down the performance of the query and of the other concurrent operations in the database system.

To illustrate how schema normalization can impact the cost of data modification operations, consider a simple schema as shown in the following example. It shows two schemas that store the same information about students in the database. In the first schema, only one table represents Student, but in schema-2, it has been broken into two tables—Student and Dept.

Schema-1:

```
Student (
    student_id          int,
    student_name        varchar(100),
    student_status      int,
    dept_name           varchar(100)
    dept_bldg_no       int,
    dept_phone          char(10))
```

Schema-2:

```
Student (
    student_id          int,
    student_name        varchar(100),
    student_status      int,
    dept_id             int)

Dept (
    dept_id             int,
    dept_name           varchar(100),
    dept_bldg_no       int,
    dept_phone          char(10))
```

Let us consider the performance of the following two operations under each of the two schemas:

- To project (for example, Select) student and department information In schema-1, you can get this information by simply scanning the table Student without requiring any joins. In schema-2, you will need to join the Student and Dept. tables.
- To update the telephone number for a department In schema-1, you will potentially need to update multiple rows in the Student table which requires more locks and log records. In schema-2, you will need to update only one row in the Dept table.

Because of data redundancy, the size of the database for schema-1 will be larger than schema-2, which can also impact the performance of your application. So which schema should be chosen by the application? The answer actually lies with the type of operations done by the application. For example, if the application is predominantly reading data (for example, a data warehouse application), then schema-1 may be okay.

The second factor is the availability of the useful indexes. A useful index is defined to be an index that can be chosen by the optimizer to process a Data Manipulation Language (DML) statement in your application. Absence of a useful index may cause a SQL Server to scan the full table, which can negatively impact the performance of the statement. For example, if there is an index on column student_name in the schema-1 above, SQL Server can locate the student by his or her name without resorting to a full-table scan. On the other hand, if the index is not useful but it exists, it can add unnecessary overhead for data modification operations, as the index entries need to be maintained to account for any changes in the table. You can take the corrective action by identifying indexes that are useful, adding the ones that are missing, and by dropping the ones that are not useful.

The third factor is the mapping of tables and databases to the physical disks or Logical Unit Numbers (LUNs), as in a Storage Area Network (SAN) environment. For the discussion here, we will use the term physical disk, but it is similar to LUNs in a SAN environment. You need to understand the access pattern of the data and then distribute the databases and tables among physical disks so as to maximize the use of available I/O bandwidth. Some examples of nonoptimal mappings are explored here.

Mixing log and data onto the same physical disk

Most operations to log files are sequential, whereas the access to data is inherently random, depending upon user queries and data modification operations. Log records are written sequentially and are accessed sequentially when they are backed up. The only other time you need to access log records in reverse order is when rolling back a transaction, which is not a common operation. Note that in SQL Server 2005, triggers are implemented using row versioning. In previous versions of SQL Server, triggers were implemented by scanning the log records in reverse order, which can cause the disk head to move back and forth, thereby impacting the I/O throughput of the physical disk containing log files. A general recommendation is to put log files onto their own physical disk. If this is not possible, you should focus on ensuring that healthy latency is maintained on log writes, as this directly impacts the response time.

Sharing physical disk(s) among and user databases

The tempdb is a shared database, one per instance of SQL Server, that is used both by the application and SQL Server to store/manipulate temporary data. You need to make sure that the tempdb is created on the disk I/O subsystem that supports the I/O bandwidth needed for the workload. If you create tempdb on a slower disk, it can severely impact the performance of the application. A general recommendation is to create tempdb on its own physical disk(s).

Mapping heavily accessed tables onto the same physical disk

A general recommendation here is to distribute I/O load across multiple physical disk(s). So in this case, you may want to consider mapping these tables to different physical disks to reduce the likelihood of an I/O bottleneck.

Transactions and Isolation Levels

Applications interact with SQL Server using one or more transactions. A transaction can be started explicitly by executing the BEGIN TRANSACTION Transact-SQL statement or implicitly, by SQL Server, for each statement that is not explicitly encapsulated by the transaction. Transactions are very fundamental to database systems. A transaction represents a unit of work that provides the following four fundamental properties:

- Atomicity Changes done under a transaction that are either all commit or are rolled back. No partial changes are allowed. It is an all-or-nothing proposition.
- Consistency Changes done under a transaction database from one consistent state to another. A transaction takes a database from one consistent state to another.
- Isolation Changes done by a transaction are isolated from other concurrent transactions until the transaction commits.
- Durability Changes done by committed transactions are permanent.

SQL server uses locks to implement transaction isolation. It acquires an exclusive (X) lock on the data before modifying it. An exclusive lock guarantees that two transactions cannot modify the data at the same time. The exclusive lock is held for the duration of the transaction so that concurrent transactions can read or modify the data only after the first transaction commits or rolls back. Similarly, a transaction acquires a shared (S) lock

before reading the data. As the name share lock implies, it allows multiple transactions to share a resource concurrently. In other words, concurrent transactions can read the same data by acquiring a shared lock of their own without blocking each other. [Table 1-1](#) shows a simple lock compatibility table. It shows that two shared locks don't conflict with each other, but an exclusive lock conflicts both with shared and exclusive locks.

Table 1-1. A Simple Lock Compatibility Table

Lock Mode	shared (S)	exclusive (X)	shared (S)	OK	N	O	Exclusive (X)	N	O
shared (S)	X	X	X	X	X	X	X	X	X
exclusive (X)	X	X	X	X	X	X	X	X	X
shared (S)	X	X	X	X	X	X	X	X	X
OK	X	X	X	X	X	X	X	X	X
N	X	X	X	X	X	X	X	X	X
O	X	X	X	X	X	X	X	X	X

As you can imagine, blocking among transactions can impact the performance of your application significantly, but it is necessary to guarantee transaction isolation property as described earlier. For this reason, the SQL Server provides

- Many more locking modes (for example, intent locks) at various granularities (for example, row, page, table) to maximize concurrency.
- Locking hints so that applications can control the locking behavior at an object level.
- Optimized code path so that acquiring/releasing locks is efficient.

As an application designer, you need to pay particular attention to minimize blocking. One interesting aspect of the isolation guarantee is the level of isolations. The SQL-99 standard defines four isolation levels that can be used to run a transaction. The transaction isolation levels provide applications with an option to choose between consistency and concurrency. The higher the consistency, the lower the concurrency. A brief description of these isolation levels follows. [Chapter 6](#), "Concurrency Problems," covers isolation levels in more detail.

- **READ UNCOMMITTED** A transaction executing under this isolation level does not need to acquire locks to read user data. This means that a transaction can read the data that is potentially modified by an active concurrent transaction without getting blocked. Since we won't know the state or the fate of the transaction that has modified the data, the application design using this isolation level must be resilient to this fact.
- **READ COMMITTED** This is the default isolation level in SQL Server. A transaction executing under this isolation level can only read the committed data and will get blocked if a concurrent transaction is modifying the data. This is accomplished by requesting the S lock on the data before reading it. However, SQL Server releases the S lock after the data has been read and does not hold it for the duration of the transaction. This allows a concurrent transaction to modify the data, even though the first transaction did not commit yet. Since the S lock is not held for the duration of the transaction, if the same data is read again, there is no guarantee that it will be the same. For example, it may have been changed by other concurrent transactions that started after the first read and committed before the second read. In other words, there is no guarantee that the read is repeatable.
- **REPEATABLE READ** This isolation level guarantees that the data read by the transaction will not change for its duration. SQL Server accomplishes this by holding the S lock for the duration of the transaction. As you can see, this isolation level offers a higher consistency to your application, but it does so at the cost of concurrency (in other words, more blocking). Now, if a concurrent transaction wants to modify the data, it has to wait for the first transaction to complete. While this isolation level guarantees repeatable read, it does not prevent phantoms. For example, if a new row gets inserted by a concurrent transaction and that qualifies the search criteria of the first transaction, this new row will show up in the result set when queried again once the transaction that inserted the new row commits.
- **SERIALIZABLE** This is the highest level of consistency offered by isolation levels. A transaction running at this isolation level reads only the committed data that does not change for the duration of the transaction, and there are no phantoms. For the purist, a serializable transaction represents a serial sequence of execution of concurrent transactions. SQL Server implements it either using key-range locks or by acquiring locks at a higher granularity, for example at table level. A transaction running at this isolation level causes the most blocking.

One thing to keep in mind is that a transaction acquires an X lock when modifying data, and this lock is held for the duration of the transaction, regardless of the isolation level of the transaction.

When designing an application you need to be cognoscente of the amount of time that the transactions will stay open. The duration of the transaction determines the duration of X locks, as these locks are held until the transaction is finished. For S locks, the transaction isolation level determines the duration of the lock and its granularity. If your application is running transactions at an isolation level higher than the default READ COMMITTED, your S locks will be held until the transaction is finished. The longer a lock is held, the higher the chances that it will block concurrent transactions, thereby impacting the throughput and the response time of your application.

At first glance, it may seem that most applications should use an isolation level that offers the most consistency. But in practice, the commonly used isolation level is READ COMMITTED and, in fact, this is the default isolation provided by SQL Server. A higher isolation level should only be chosen after careful consideration.

Let us consider a simple hotel reservation application and explore couple of ways to implement it using different transaction isolation levels.

Assume your room reservation application is a Web-based application that displays room information, including availability and the nightly rate. This application allows online customers to reserve a room and also allows hotel staff to change the nightly rate of the room depending upon the season and current vacancy rate. Let us assume that there is one row per room per day of the year. One of the main design considerations is to prevent two customers from reserving the same room. We'll look at two possible ways to implement a solution.

First Implementation

A user searches for a room meeting his/her criteria. The application queries the database and displays the rooms that satisfy the criteria. To guarantee that the information about the room(s) does not change, the transaction is run at a repeatable read isolation level that obtains S lock(s) on the row(s) representing the room(s). The user decides to reserve one room, provides the payment information, and then submits the Web-form. Because an S lock was held on the row representing the room, no one else would have reserved this room, the application updates the reservation status of the room, and then commits the transaction. The customer is informed that the reservation was successful.

There are a few issues with this implementation. First, the transaction is active while waiting for the customer to browse and update the room information. This means that the transaction duration is unlimited, as it depends upon when the customer submits the form. A long-running transaction means that the lock acquired will be held for the longer duration, as well leading to more blocking if there are concurrent transactions that need to modify this data. Second, if two customers browsing the same room want to reserve it, a deadlock will occur because the transactions representing the customers will hold the S lock on the row, and now both want to acquire X lock on it. The customers will end up waiting for each other leading to a deadlock. You can eliminate deadlock by doing the SELECT with UPDLOCK hint, but you will still have blocking.

Second Implementation

A user interacts with the application as in the previous implementation, but the application is in the READ COMMITTED isolation level and returns the room meeting the search criteria without holding any locks and completes the transaction. However, to prevent concurrent reservations by multiple customers, a new column timestamp is added to the schema that tracks the time of the last update. So when a customer submits the room reservation request, the application verifies if the timestamp of the row to be updated is still the same. If it is, the room reservation is successful; otherwise, it fails and the customer is given the message "Sorry, the room

is already taken." The interesting part about this implementation is that it avoids lost-update (that is, multiple customers reserving the same room) while running the transactions for a shorter duration at a lower isolation level. So it is a preferred design over the first implementation.

This example illustrates one of many poor application designs that can lead to unnecessary blocking. If you see a lot of blocking in your application, you will need to pay close attention to the duration of transactions and the isolation level used. In SQL Server 2005, you may also consider using SNAPSHOT ISOLATION or READ_COMMITTED_SNAPSHOT to remove blocking between readers and writers (for example, SELECT and UPDATE). [Chapter 6](#) covers isolation levels in more detail.

Transact-SQL Code

Though the SQL Server optimizer does rearrange Transact-SQL submitted for better optimization, it cannot compensate for a poor design. One such example is using the cursor to update multiple rows. Because the cursor operates on one row at time, it will be significantly slower than the corresponding set operation. Here is one example to illustrate this point.

Code View:

```

CREATE TABLE t1 (c1 int PRIMARY KEY, c2 int, c3 char(8000))
GO
â   Load 6000 rows into this table as follows
DECLARE @i int
SELECT @i = 0
WHILE (@i < 6000)
BEGIN
    INSERT INTO t1 VALUES (@i, @i + 1000, 'hello')
    SET @i = @i + 1
END

â   Method-1: Update all the rows in this table in a single statement

BEGIN TRAN
UPDATE t1 SET c2 = 1000 + c2
COMMIT TRAN

â   Method-2: Update all the rows in this table using cursor
DECLARE mycursor CURSOR FOR
    SELECT c2 FROM t1
OPEN mycursor
GO
BEGIN TRAN
FETCH mycursor
WHILE (@@FETCH_STATUS = 0)
BEGIN
    UPDATE t1 SET c2 = 1000 + c2 WHERE CURRENT OF mycursor
    FETCH mycursor
END
COMMIT TRAN
â   Now query the total worker time
SELECT TOP 10
    total_worker_time/execution_count AS avg_cpu_cost,
    execution_count,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(max), text)) * 2
            ELSE statement_end_offset
        END - statement_start_offset)/2)
    FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM sys.dm_exec_query_stats
ORDER BY [avg_cpu_cost] DESC

```

â here is the output of the DMV query

avg_cpu_cost	exec_count	query_text
4913844	1	UPDATE [t1] SET [c2] = @1+[c2]
58634	1	SELECT top 10 total_worker_tim
8437	6000	FETCH mycursor
5641	1	FETCH mycursor
5268	6000	UPDATE t1 SET c2 = 1000 + c2 w

In this case, the updates using the cursor are taking $(8,437 * 6,000 + 5,268 * 6,000) = 81,690,000$ units. This is a factor of approximately 20, compared to 4,913,844 units taken by set-oriented update. It is unlikely that an application will use a cursor to update a large set of rows when it can alternatively use a set-oriented Transact-SQL UPDATE statement as described previously. However, the point is that poorly designed Transact-SQL can lead to performance issues in your application, and it is worth examining.

Hardware Resources

Hardware resources are the horsepower you need to run your application. It will do you no good if you have a well-designed application, but it is running on hardware that is not on a par with the demands of the workload. Most often, an application is tested in a small-scale environment with simulated workload. While this does serve a useful purpose, clearly it is not a replacement for measuring the performance of the workload and the hardware it actually runs on in production. A study by the Morse Consultancy Group (<http://www.morse.co.uk>) indicates that 89 percent of the organizations have inadequate processes and hardware for performance testing. So when your application hits performance issues, it can be the hardware, however, that is not necessarily so. Any hardware resource (CPU, I/O, memory, or network) that is pushed beyond its operational capacity in any application tier will lead to a slowdown in your application. Note, too, that hardware bottlenecks may also be caused by poor application designs, which ultimately need to be addressed. In such cases, upgrading hardware is a short-term solution at best.

SQL Server Configuration

SQL Server is designed to be self-tuning to meet the challenges of the workload. In most cases, it just works well with out-of-the-box configuration settings. However, in some cases, you may need to tweak configuration parameters for maximum performance. Inside SQL Server 2005: The Storage Engine (Microsoft Press, 2006) has some additional details about configuration options. In this section, we will list the options that you will most likely need to monitor to track down performance issues. The relevant configuration options can be considered to be either CPU-related or memory-related options.

CPU-Related Configuration Options

The CPU-related configuration options are used, for example, to control the number of CPUs or sockets that can be used by a SQL Server instance, maximum degree of parallelism, and the number of workers. Some commonly used options in this category include the following:

- Affinity Mask This option can be used to control the mapping of CPUs to the SQL Server process. By default, a SQL Server uses all processors available on the Server box. A general recommendation is not to use affinity mask option, as SQL Server performs best in the default setting. You may, however, want to use this option under two situations.

- ◆ First, if you are running other applications on the box, the Windows operating system may move around process threads to different CPUs under heavy load. By using affinity masks, you can bind each SQL Server scheduler to its own CPU. This can improve performance by eliminating thread migration across processors, thereby reducing context switching.
- ◆ Second, you can use this parameter to limit the number of CPUs on which a SQL Server can run. This is useful if you are running multiple SQL Server instances on the same Server box and want to limit CPU resources taken by each SQL Server and to minimize their interference.
- Lightweight Pooling When this configuration is enabled, SQL Server makes use of Windows fibers. A worker can map to a Windows thread or to a fiber. A fiber is like a thread, but it is cheaper than normal thread because switching between two workers (that is, fiber threads) can be done in user mode instead of kernel mode. So if your workload is experiencing a CPU bottleneck with significant time spent in kernel mode and in context switching, you may benefit by enabling this option. You must test your workload with this option before enabling it in the production system, as more often than not this option may cause performance regression. You should also keep in mind that CLR integration is not supported under lightweight pooling.
- Max Worker Threads You can think of workers as the SQL Server threads that execute user or batch requests. A worker is bound to a batch until it completes. So the maximum number of workers limits the number of batches that can be executed concurrently. By default, SQL Server sets the Max Worker Threads as described in the following table.

Number of CPUs	32-bit computer	64-bit computer
<= 4 processors	256	512
8 processors	288	576
16 processors	352	704
32 processors	480	960

For most installations, this default setting is fine. Each worker takes 512-KB memory on a 32-bit, 2 MB on X64, and 4 MB on IA64. To preserve memory, the SQL Server starts off with a smaller number of workers. The pool of workers grows or shrinks based on the demand. You may want to change this configuration under two conditions. The first is if you know that your application uses a smaller number of workers. By configuring it to a lower number, the SQL Server does not need to reserve memory for the maximum number of workers. The second is if you have many long-running batches (presumably involving lock waits), such that the number of workers needed may exceed the default configuration. This is not a common case, and you may want to look at your application design to analyze this.

- Max Degree of Parallelism This configuration parameter controls the maximum number of processors or cores that can be deployed to execute a query in parallel. Parallel queries provide better response time but take more CPU resources. You need to look into this configuration parameter if you are encountering CPU bottleneck, described later in this chapter.

Memory-Related Configuration Options

The memory-related configuration options are used to control the memory consumed by SQL Server. Some of the commonly used configuration options in this category include the following:

- Max and Min Server Memory This is perhaps the most critical of the configuration options, especially in 32-bit configurations, from a performance perspective. This configuration parameter is often confused with the total memory configured for a SQL Server, but it is not the same. It represents the configured memory for the buffer pool. SQL Server, or any database server for that matter, is a

memory-hungry application. You want to make as much memory available for SQL Server as possible. The recommendation on 64 bits is to put an upper limit in place to reserve memory for the OS and allocations that come from outside the Buffer Pool. You will also need to cap the memory usage by SQL Server when other applications (including other instances of SQL Server) are running on the same Server box.

- **AWE Enabled** On a 32-bit box, the SQL Server process can only address 2 GB of virtual memory, or 3 GB if you have added the /3 GB parameter to the boot.ini file and rebooted the computer, allowing the /3 GB parameter to take effect. If you have physical memory greater than 4 GB and you have enabled this configuration option, the SQL Server process can make use of memory up to 64 GB normally, and up to 16 GB if you have used /3 GB parameter. Additionally, the SQL Server process requires Lock Pages in Memory privilege in conjunction with AWE-Enabled option. There are some restrictions in terms of the SQL Server SKU and the version of the Windows operating system under which you are running. Please refer to SQL Server 2005 Books Online (BOL) onTechNet for more details. You should be aware of certain key things when using the AWE option. First, though it allows the SQL Server process to access up to 64 GB of memory for the buffer pool; the memory available to query plans, connections, locks, and other critical structures is still limited to less than 2 GB. Second, the AWE-mapped memory is nonpageable and can cause memory starvation to other applications running on the same Server box. Starting with SQL Server 2005, the AWE memory can be released dynamically, but it still cannot be paged out. SQL Server may release this memory in response to physical memory pressure. Third, this option is not available on a 64-bit environment. However, if the SQL Server process has been granted Lock Pages in Memory privilege, the buffer pool will lock pages in memory and these pages cannot be paged out.

So far, we have discussed some of the major considerations when designing and deploying a database application, particularly in the context of SQL Server, though most of the discussion is also applicable to other database servers as well.

Troubleshooting Overview

In a perfect world, your database application is well designed; the hardware configuration that it is running on meets or exceeds the demands of the workload; you have full control on the environment; applications and users interact with SQL Server using well-formed Transact-SQL queries and don't make mistakes. However, that is rarely the case. A large numbers of IT shops run independent software vendor (ISV) applications and have almost no control over its design. Over time, the workload outgrows the hardware capacity. Many applications may run on the same hardware and may interfere with SQL Server. Workload may change drastically over a short period of time. And finally, a user mistake, like dropping an index, may cause performance regressions. As a DBA or system administrator, you are faced with these challenges on a daily basis. What is your recourse? How do you identify these performance problems proactively and troubleshoot them in a timely manner? Is the problem in the application or the hardware or the workload or user action? The possibilities are many and without a well-thought-out strategy, you will end up wasting a lot of time. Broadly speaking, some of the key elements of a good strategy for performance troubleshooting are:

- Creating a baseline for your workload.
- Monitoring your workload.
- Detecting, isolating, and troubleshooting performance problems.

In the sections that follow, we describe each of these in detail. Because the problem space is huge and sometimes depends on the unique issues at a given installation, it is not possible to provide a universal strategy that is applicable for all situations. However, we believe that this strategy can be applied to most commonly occurring performance problems.

Creating a Baseline for Your Workload

For most customers, the performance troubleshooting is more reactive than proactive. When the application users start experiencing performance problems, the system or database administrator typically looks at the performance counters to get some clues. The number of performance counters is large, therefore most of these counters have a huge operating range, and it is nearly impossible to say if the value of a specific counter is reasonable or not. For example, how do you know if the SQLServer:Locks:lock waits/Sec that you are seeing is within acceptable limits? It is normal to expect some lock waits in any application that allows concurrent access to the data, but the question is what is reasonable for your application at the current workload on the specific hardware that you are running? For any performance counter, multiple dimensions impact its value, which complicates the analysis further. So what should you do? Well, the short answer is that you need to have a baseline with which to compare (that is, a baseline of performance counters when your application is running normally). In fact, you need to have a baseline of any piece of data, not necessarily the performance counters, that you can use to debug performance problems. So for example, you may want to have a baseline of optimized query plans for key queries in your applications. Baseline allows you to compare the current performance data collected with the data from when the application did not have any performance problems. Using this information, you can quickly spot the numbers that are not within the normal operating range and can use this information to narrow down the cause of performance issue(s). There are many reasons why you see a sudden performance slowdown. A couple common ones are as follows:

- A useful index is dropped accidentally This may lead to an expensive query plan that requires a table scan causing memory and I/O pressure. If your baseline has the query plan for the queries that are affected, you can easily identify the cause.
- Unplanned changes in the workload An example of this sort might be running reports at the end of the month or seasonal sales activity like the day after Thanksgiving and so on. While a slight degradation in the response time may be expected if the hardware your SQL Server is running on is not designed to handle such changes, users will experience a slowdown. In this case, you will see pressure on hardware resources like CPU utilization near 100 percent or the memory pressure that leads to more paging, which, in turn, affects I/O. Interestingly, the symptoms (for example, pressure on hardware resources) here may be similar to the previous case when an index was dropped accidentally, but the cause is completely different. This is as good a place as any to emphasize that throwing hardware at the performance problem is not always the best solution. Again, by comparing your performance data with the baseline, you will be able to narrow down the cause of the performance issue.

Baseline is an essential part of any performance troubleshooting methodology. For most applications, one baseline may not be sufficient, and you may require multiple baselines. For example, consider a stock-trading application. The workload on this application will vary depending on the time of the day; more like Online Transaction Processing (OLTP) load during trading hours and Data Warehouse (DW)â like workload to generate reports during post-trading hours. Similarly, the workload could be different at the end of quarter or at the end of the financial year. This brings up two key points. First, you need to baseline your workload to identify trouble spots quickly. Second, you may need multiple baselines, each representing different types of workloads in your application usage. When you have multiple baselines, you will need to use the corresponding or representative baseline when troubleshooting performance problems in your application.

We hope that now you may have some appreciation of why is it important to baseline your workload, but one question remains and that is "What should I baseline?" This is a harder question to answer than you might initially believe. Clearly, a generic answer is that we want to baseline data that will help us identify the performance problemâ but what exactly is that? This issue is further complicated by the fact that there are many tools available that can be used to monitor performance of your SQL Server. If you use every available tool and use the full set of information they expose, very soon you will have huge amount of data to collect and analyze. Also, running all the tools will impact the performance of your workload. You need to strive to find a good balance between collecting enough information and limiting the impact on the workload. Let's look at some of the commonly used tools for measuring workload and establishing a baseline.

System Monitor

This tool is also called Performance Monitor, and is a Windows monitoring tool that provides usage information on CPU, memory, and disk and network resources, among many other useful counters. This tool provides information at the process level for the application running on Windows platform. However, applications running on the Windows operating system can be integrated with this tool to expose useful information inside the application process itself. As you can imagine, the SQL Server application is integrated with System Monitor, and it provides many useful counters related to SQL Server. Some common ones are Page Life Expectancy, Buffer cache hit ratio, Free Space in tempdb (KB), and more. The overhead of monitoring the counters will depend on number of counters and the frequency of monitoring. System Monitor allows you to monitor as frequently as every second.

SQL Server Profiler

The SQL Server Profiler is a graphical user interface (GUI) to Microsoft SQL Trace for monitoring an instance of the Microsoft SQL Server Database Engine. It can capture the data about the events of interest. This data can be saved to a file or to a table for later analysis. It is generally recommended to collect data in a file instead of a SQL Server table, as depending on the events being monitored, this table can have large number of inserts, thereby taxing precious resources available to run the application. It is also recommended that you create the file on a local disk subsystem not shared with your SQL Server datafiles or executables. Unlike System Monitor, which is based on polling, the SQL Server Profiler is event-based. With a polling system, you can potentially miss information you are monitoring if an event happens to occur in the middle of a polling interval. This doesn't mean that no events will be missed using the SQL Server Profiler. On a heavily loaded system, it is recommended that you create a server-side trace and send your trace data directly to a server-side file. Using a server side-trace will ensure that no events are dropped, while events may be dropped when sending events to a Profiler defined trace, if the system is very busy. SQL Profiler is expensive to run. Its impact on the performance of your workload will depend on the events selected and the frequency at which those events are generated. So server-side traces are preferable not just to avoid dropping events, but to minimize the impact on your SQL Server. A typical usage scenario for SQL Profiler is to collect event data at short intervals and then correlate events with executing queries to identify the problem. SQL Server Tracing and the SQL Server Profiler are discussed in much greater detail in [Chapter 3](#).

Database Engine Tuning Advisor (DTA)

The Database Engine Tuning Advisor (DTA) is a more powerful replacement for Microsoft SQL Server 2000's Index Tuning Wizard (ITW). It is used to analyze the typical application workload collected using SQL Profiler and the physical schema to come up with recommendations for useful indexes. Besides recommending useful indexes, it provides recommendations on partitioning, online operations, and many other situations. This is a good tool when you are initially setting up your application because it analyzes the application statically using the SQL Profiler trace. However, it has limited use once the system is operational. You may need to run this tool again if your application has changed, for example, if you installed a newer version of the application or if your workload has changed significantly.

DBCC Commands

DBCC stands for Database Console Commands. Most DBCC commands are related to checking the consistency of the database. One example is DBCC CheckDB command. However, there are some DBCC commands like DBCC Memorystatus that can be used to monitor memory usage in SQL Server or DBCC SQLPerf that can be used to monitor transaction log space usage. Since DBCC is not the right tool to expose performance-related information, starting with SQL Server 2005 the new Dynamic Management Views and Functions, as described next, are the preferred ways to expose such information.

Dynamic Management Views (DMV) and Functions (DMF)

These views and functions expose internal data structure of SQL Server and the associated information that is relevant for diagnosing and troubleshooting performance problems. Since DMVs and DMFs are very similar, in the text that follows, we will use the term DMV to refer to both DMFs and DMVs.

DMVs expose this information in relational format so that it can be accessed using familiar paradigm, using Select statement, and can be combined with other DMVs using join operator(s) to provide meaningful data. Since most information exposed needs to be maintained by SQL Server anyway, there is no additional overhead of collecting this information inside SQL Server. However, there is a small overhead when you use DMVs to retrieve this information as the in-memory structures need to be presented as a relational rowset. Similarly, if you use a complex query involving multiple DMVs, the cost of query execution will be proportional. Still, the overhead of running DMVs is quite low compared to SQL profiler. It primarily involves CPU overhead of less than 2 percent for most DMVs^[1] and no additional physical or logical I/Os, except in few cases. Besides, DMVs expose much more useful information than currently possible with SQL Profiler. However, like System Monitor counters, the information retrieved through DMVs is based on polling and it is possible to miss useful information. For example, the DMV sys.dm_os_waiting_tasks tracks the tasks that are currently blocked. So it is possible that we may miss the task(s) that became blocked and then unblocked between the polling intervals.

^[1] Some DMVs like sys.dm_tran_version_store or sys.dm_os_buffer_descriptors are expensive to run, but you don't typically use these for performance monitoring.

It is interesting to observe that the concept of DMVs is not new in SQL Server 2005. You may be familiar, for example, with sysprocesses system table that returns the information about active sessions and the requests they are executing in a relational rowset format. So this is in fact a DMV. Similarly, there are some stored procedures that expose internal information in relational rowset format. For example, sp_lock that exposes locking related information in relational rowset format. The key change in SQL Server 2005 is that the DMVs are now formalized and that there are lot more DMVs. However, with this change comes the challenge of how to use the information exposed by DMVs effectively. We recognize this, so where possible, we will provide examples using DMVs in the context of the performance problem at hand.

We consider DMVs and System Monitor Counters to be very powerful tools at our disposal, and, as you will see in later sections, we use them extensively in troubleshooting performance problems.

Monitoring the Workload

What good is a baseline if you don't monitor your workload regularly to catch significant deviations from the baseline(s)? Any significant deviation from baseline represents a change that needs to be understood and analyzed for its impact on the performance of your workload. There can be many reasons why the performance of your workload drops. Some common causes in the context of SQL Server are described here.

- Query plan is altered because of the changes in statistics or a drop of a useful index. The modified plan may take more resources, thereby causing a general slowdown.
- Change in the locking granularity. SQL Server uses a complex heuristic to choose an appropriate locking granularity considering the selectivity of the predicate and the size of the table and the concurrent workload. If it chooses a coarser locking granularity (for example, Table lock), it may cause significant blocking, and if it chooses lower granularity (for example, row), it will increase the locking overhead.
- Rogue session. A user may forget to add a predicate to a query, which then leads to a table scan of a huge table. Similarly, a user may execute a complex query and depending on the complexity (for example, the number of joins) of the query, it can be huge drain on the system resources.
- Change in the workload. For example, if you are a news content provider, an international event may suddenly flood users to your Web site, causing significant delays.

- Normal growth of business and related workload By monitoring your workload, you can identify the trend and use this information to plan for changes.
- Upgrade to a newer version of SQL Server For most applications, the newer version of SQL Server will help improve its performance, but still you need to monitor your application with the newer version of SQL Server and identify if there are some deviations from the baseline.
- Hardware resources If a new application or software is run on the same box as SQL Server, it will compete for hardware resources, which may lead to the SQL Server process not getting enough CPU or memory for the workload.

Most performance problems ultimately manifest themselves into resource bottlenecks. Monitoring your workload periodically enables you to identify the resource bottlenecks early. Monitoring in the context of DMVs/System Monitor counters means that you need to poll these at regular intervals. You may wonder how often you should monitor. It really depends on how dynamically changing is your workload and what monitoring overhead are you willing to accept. If you monitor (i.e., poll) frequently and monitor large numbers of System Monitor counters and DMVs, you will also need to worry about where to store this information and how to process it. We will touch upon this later in this chapter.

Generally speaking, the resource bottlenecks should be treated as symptoms and not necessarily the cause of the performance problem, though they can very well be. The troubleshooting strategy that we follow has three steps for each type of resource. First is the detection that we, indeed, have a resource bottleneck. Second is the isolation of the cause(s) of the resource bottleneck. The third and final one is the resolution of the performance problem by taking corrective actions.

Detecting, Isolating, and Troubleshooting Common Performance Problems

We realize that it is nearly impossible to touch upon all performance problems encountered by millions of customers, who have deployed SQL Server to run their database applications. We will focus on troubleshooting some of the common performance problems that we have encountered frequently. However, we do think the steps you take to identify and troubleshoot performance are very similar and are relevant for most database applications.

When an application experiences a performance problem, it invariably bottlenecks on some resource be it CPU, memory, I/O, or network bandwidth. Additionally, in the context of SQL Server, it can be caused by excessive locks (that is, blocking) and also by contention in one special resource called tempdb. Keep in mind that tempdb is a shared resource across all databases in a SQL Server instance and any contention or space issues in tempdb can potentially impact all applications running on the SQL Server instance. For the discussion here, we will focus on the bottlenecks in following resources:

- CPU
- Memory
- I/O
- tempdb
- Blocking

For each of these resources, we will discuss how to detect that there is a bottleneck, how to isolate the causes, and how to troubleshoot and alleviate the bottleneck.

CPU Bottlenecks

Before we discuss CPU bottlenecks, let us go over some basic concepts for better understanding of the execution model inside SQL Server.

The SQL Server process has a set or pool of workers that are used to execute user queries and internal background tasks like lazywriter, which moves the dirty pages to the disk. A worker represents a logical thread in SQL Server that is internally mapped (1:1) to either a Windows thread or, if lightweight pooling is turned ON, to a fiber. A user or an application submits a query or a group of Transact-SQL statements to SQL Server for execution. This unit of work is called a batch or a request. When the SQL Server receives a batch, it assigns a worker to it for execution. The association of a worker to the batch is kept for the life of the batch. This association is kept, even if the worker is blocked on lock request or on an I/O. Once the batch is completed, the worker is then free to execute another batch. Note, in case the SQL Server decides to execute a batch in parallel, it will assign multiple workers to execute it. If concurrent user requests/batches are submitted, SQL Server will associate a worker with each of the batches and the worker(s) will then execute the batch to completion. As you can guess, the number of active workers (in other words, the workers that are executing a batch) is an indication of the CPU load on SQL Server. But what if all workers are blocked on resource? In that case, you will find that even though we have a large number of active workers, the CPU overhead may be very low. This brings up an interesting point that we may need to look at multiple pieces of information before drawing any conclusion about almost any performance issue.

A worker can be in many states, but the interesting states are RUNNING, RUNNABLE, and SUSPENDED:

- RUNNING The worker is currently executing on the CPU.
- RUNNABLE The worker is currently waiting for its turn on the CPU.
- SUSPENDED The worker is waiting on a resource, for example, a lock or an I/O.

So if you have a large number of workers in RUNNABLE state, it is symptom of CPU bottleneck. On the other hand, if your workers are spending most time in SUSPENDED state, it is indicative of excessive blocking in your SQL Server.

Detection

You can identify a CPU bottleneck by looking at System Monitor counter Processor:% Processor Time. If the value of this counter is high, say greater than 80 percent, a generally accepted number, for 15 to 20 minutes, it indicates that you have CPU bottleneck. You can also monitor System:Processor Queue Length. A sustained value of 2 or higher typically indicates CPU pressure. However, this does not necessarily mean that the CPU bottleneck is caused by an actual SQL Server process as there may be other applications running on the same box. Unless SQL Server is the only application running out of the box, you can find out the CPU resources taken by SQL Server process by looking at the process-specific counter Process:%Processor Time. If you find that it is the other application that is causing the CPU bottleneck, then clearly it is futile to look inside SQL Server for solutions. If you need to share your SQL Server box with other applications, then how do you know what the right CPU percent is for SQL Server or other processes? This is where the baseline is your friend. At any time, you can compare the workload on your box with your baseline and can isolate which process is over its operational range.

Another way to detect CPU pressure is by counting the number of workers in the RUNNABLE state. You can get this information by executing the following DMV query:

```
SELECT COUNT(*) AS workers_waiting_for_cpu, t2.Scheduler_id
FROM sys.dm_os_workers AS t1, sys.dm_osSchedulers AS t2
WHERE t1.state = 'RUNNABLE' AND
      t1.scheduler_address = t2.scheduler_address AND
      t2.scheduler_id < 255
GROUP BY t2.scheduler_id
```

You can also use the time spent by workers in RUNNABLE state by executing the following query:

```
SELECT SUM(signal_wait_time_ms)
FROM sys.dm_os_wait_stats
```

Signal waits represent the difference between the time a worker entered the RUNNABLE state and the time when it actually started running. A significant deviation from your baseline indicates some changes in the workload that is causing CPU pressure. One thing to keep in mind is that the above DMV returns waits since the SQL Server was started. So to get meaningful data, you will need to look at the delta between the periods of interest.

Isolating and troubleshooting of CPU bottlenecks

Once you know that the slowdown in your application is caused by a CPU bottleneck, you need to isolate the cause. A CPU bottleneck may be caused by multiple factors, but some common reasons are as follows:

An inefficient query plan

A good starting point here is to identify queries that are taking the most CPU time and compare them with the time taken in your baseline. If you see a significant deviation, you will need to take a look at that query(s) to see what could have caused it. It is possible that the updates in the statistical information caused the query plan to change. SQL Server Optimizer is cost-based, and it relies heavily on the statistical information to compute the size of intermediate result sets that may determine the join order and/or the join strategy. Any significant updates to statistical information can cause the query plan to change and sometimes it may change for the worse. Similarly, if a useful index was dropped by mistake, the query plans that depended on this index would need to be recompiled and the resultant query plan may become expensive.

Here is a DMV query that can be used to get the top-10 queries that are taking the most CPU per execution. It also lists the SQL statement, its query plan, and the number of times this plan was executed. If an expensive query is executed very infrequently, it may not be of that much concern.

```
SELECT TOP 10
    total_worker_time/execution_count AS avg_cpu_cost, plan_handle,
    execution_count,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(max), text)) * 2
            ELSE statement_end_offset
        END - statement_start_offset)/2)
    FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM sys.dm_exec_query_stats
ORDER BY [avg_cpu_cost] DESC
```

Note, this DMV only shows the aggregated statistical information for the queries that are currently cached. It is possible that some expensive queries may have been removed from the cache because of memory pressure. However, if you are polling this DMV at regular intervals, you have already likely captured those queries. Comparing the avg_cpu_cost of the queries with the corresponding baseline can provide you an immediate starting point from where to look for the potential problem. You will need to investigate each of these query plans and identify if a modified query plan will help reduce its cost of execution. Also, you should look at whether you can simplify the Transact-SQL associated with these plans, which may reduce its execution cost. Here is the simplified query output of the above DMV. We can make few interesting observations about the workload. First, even though the CPU cost of executing the select sum(c3) from t1 query is not significant, it has been executed 11,002 times. So it is important to look at its query plan to find opportunities to optimize it.

For example, if you create a nonclustered index on column c3, the previous query can be computed by just scanning the nonclustered index without any need to access the data pages. Second, the most expensive query involves a join. You will need to look at its query plan to see if there are ways to optimize it even further.

avg_cpu_cost	plan_handle
-----	-----
29589011	0x06000500E401FC06B801570400000000000000000000000000000000
83021	0x06000500201C8F03B8E1BB0B00000000000000000000000000000000
62786	0x06000500334B0B05B861080700000000000000000000000000000000
Execution_count	query_text
-----	-----
10	select c1, c5 from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4 order by c2
11002	select sum(c3) from t1 select count(*) from t2
1	

If you use the above DMV, you may miss finding the most frequently executed queries in your workload if the CPU cost of those queries is much less than say top-10 queries. To find the most frequently executed queries in your workload, you can execute the following DMV query, a slight variant of the previous DMV query:

```
SELECT TOP 10 total_worker_time, plan_handle, execution_count,
(SELECT SUBSTRING(text, statement_start_offset/2 + 1,
(CASE WHEN statement_end_offset = -1
      THEN LEN(CONVERT(nvarchar(max),text)) * 2
      ELSE statement_end_offset
      END - statement_start_offset)/2)
 FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM sys.dm_exec_query_stats
ORDER BY execution_count DESC
```

Excessive compilation and recompilation

When a query or a batch is submitted, it is first compiled to generate an optimized query plan, and then it is executed. The optimized plan is kept in the cache also known as the procedure cache. If the same query or batch is run again, the SQL Server first checks the procedure cache for the optimized plan of the query and, if one is found, it is reused and executed. This is particularly critical for complex queries that typically involve many join operations because in this case, the solution space that the optimizer needs to explore is exponentially larger. By reusing optimized query plans, the cost of optimization can be amortized over multiple invocation of the same query or the batch. If you are incurring lot of recompiles, you may see a higher CPU utilization because compiling or recompiling is a CPU-intensive activity. There are many reasons that SQL Server will automatically recompile a query. Some of the more common reasons are the following:

- Schema Change If the metadata of the referenced objects is changed, it causes a recompile. So if you have a batch that mixes DDL and DML, it will force a recompile.
- Set Option There are some set options which will cause a recompile, if changed. Some of these options are ANSI_NULLS, ANSI_PADDING, ANSI_NULL, and ARITHABORT. If you change these options inside a batch, it will force a recompile every time.
- Update Statistics SQL Server uses a cost-based optimizer. It uses statistical information for the tables and indexes to compute the size of the intermediate results. Any significant changes in the statistical information will force a recompile.

- Recompile Option If you have a stored procedure with recompile option, it will get recompiled on every execution. You may want to have a recompile option if the optimized plan for the stored procedure is very sensitive to parameter values.

In SQL Server 2005, the recompile is at the level of statement. So only the affected statements are recompiled, unlike SQL Server 2000, in which the whole batch is recompiled. You can use following System Monitor performance counters to see the rate of compile and recompile:

- SQLServer: SQL Statistics: Batch Requests/Sec
- SQLServer: SQL Statistics: SQL Compilations/Sec
- SQLServer: SQL Statistics: SQL Recompilations/Sec

It is important to look at compile/recompile numbers in relation to number batch requests per second. You can also compare these numbers with the corresponding baseline, if you have one, and see if we are encountering more compile/recompiles currently. Note that compilation represents the first time a query is compiled while the recompliations represent the subsequent compilation of a memory resident query plan.

It is also useful to know how much time SQL Server is spending in optimizing the query plans. You can use the following DMV query to get this information:

```
SELECT *
FROM sys.dm_exec_query_optimizer_info
WHERE counter = 'optimizations' OR
counter = 'elapsed time'
```

These counters are cumulative counters from the time the instance was started. You can find the specific number of optimizations and the elapsed time by taking two different snapshots of this DMV. The counter optimizations represent the total number of query/batch optimizations between the two snapshots and the average elapsed time for each optimization in seconds. Since query optimization is a CPU-bound activity, you can get a good idea of the CPU cost of query optimization. If the query optimization overhead is significant, you will need to identify the reasons and address them.

To identify queries/batches that are being recompiled frequently, you can, of course, use SQL Profiler to get this information. However, it is not a preferred option for reasons we explained earlier. In SQL Server 2005, you can use DMVs to find the top-10 query plans that have been recompiled the most.

```
SELECT TOP 10 plan_generation_num, execution_count,
(SELECT SUBSTRING(text, statement_start_offset/2 + 1,
(CASE WHEN statement_end_offset = -1
      THEN LEN(CONVERT(nvarchar(max),text)) * 2
      ELSE statement_end_offset
      END - statement_start_offset)/2)
 FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM sys.dm_exec_query_stats
WHERE plan_generation_num >1
ORDER BY plan_generation_num DESC
```

By looking at each of the plans and the associated SQL Text, you can possibly identify the reasons that lead to recompliations. We had already alluded to some common reasons of recompliations and you will need to take a look at those. [Chapter 5](#), Plan Caching and Recompilation, discusses compilations and recompilation in much more detail.

If you are encountering an increase in number of compilations, it is also likely that SQL Server is under memory pressure, and it is kicking out the query plans from memory. This is one classic example where a memory pressure in your SQL Server may surface as a CPU bottleneck. You must always evaluate resource bottlenecks in one resource in the context of other resources. So if you have identified that the number of compilations have increased, you must look at the memory allocated to procedure cache, a cache used to store optimized plans. You can execute the following DBCC command to get that info:

```
DBCC MEMORYSTATUS
```

Here is a sample output after removing other information. TotalPages represent stolen buffer pool pages used to store optimized plans

Procedure Cache	Value
-----	-----
TotalProcs	37
TotalPages	899
InUsePages	20

By comparing this with your baseline, you can estimate if your procedure cache size has shrunk and that may imply that SQL Server is under memory pressure. You will need to analyze why SQL Server is under memory pressure and address it. The next section provides a detailed look at detecting memory pressure and how to manage it.

Memory Bottlenecks

Most production databases are of much larger size than the memory available to database server process. As SQL Server accesses database pages to process user queries, a physical I/O occurs if the requested page is not found in the SQL Server process memory. SQL Server caches pages in memory in an internal structure known as a buffer pool and deploys a version of Least Recently Used (LRU) caching algorithm to maximize the likelihood of commonly accessed pages being found in the buffer pool. Besides storing pages, which have been read from disk, the SQL Server uses memory for internal structures like locks, connections, workers, and to store optimized query plans. So if your SQL Server has not been configured with the appropriate amount of memory, you may see a significant drop in the performance of your application. The impact of insufficient memory often spills into the bottleneck in other resources. For example, a query plan may need to be kicked out of memory under memory pressure. If the query is resubmitted for execution, it will need to be optimized again, which can put pressure on the CPU as query optimization is a CPU-intensive operation. Similarly, database pages may need to be removed from the buffer pool under memory pressure. If those pages need to be referenced soon thereafter, it will lead more physical I/Os.

Commonly, when we think of memory, we think of physical memory (that is, RAM) available on the server box. There is another kind of memory, virtual address space (VAS) or virtual memory, which is equally important. Using the Windows operating system, all 32-bit applications have a 4-gigabyte (GB) process address space that can be used to access a maximum of 4 GB of physical memory. Out of this 4 GB of total addressable memory, 2 GB of VAS is available to process to access in user-mode and the other 2 GB is reserved to be accessed in kernel mode. You can change this configuration using a /3 GB switch in boot.ini file. This switch provides applications with access to 3 GB of VAS in user mode, while limiting the VAS accessible in kernel mode to 1 GB. Though a process can access 4 GB of address space, it does not mean that you require 4 GB of physical memory. A common operating system mechanism to achieve this is called paging. It uses a swap file to store part of a process memory that was not recently referenced. When this memory is referenced again, it is transparently read back (or paged in) into the physical memory from the swap file. So if you have 4 GB of physical memory (RAM) and 8 GB of swap file, the total VAS that can be

committed is 8 GB by all active processes on the server box, but note only 4 GB of the referenced memory is actually in the RAM and the rest of the other memory is considered "paged out." For example, if you have a process that has been inactive for a while, the operating system will page it out under memory pressure. If this process is activated later, it will appear slow or unresponsive because it takes time to "page in" the memory associated with the process. Similarly, if currently running processes have committed most of the available memory, for example 8 GB in the case just described, new memory can't be allocated to the process. As you can see, there are two kinds memory pressures, the physical memory pressure and the virtual memory pressure, that can impact the performance of your application. The following section provides details on each of these.

Physical memory pressure

SQL Server can experience physical memory pressure caused by external components or from within itself. It experiences external memory pressure if there is not enough physical memory (that is, RAM) available to SQL Server. This can occur either because you have very limited physical memory on the server box or the server box is being shared with other applications including another instance(s) of SQL Server(s). In such a case, you will have a situation where its working set (that is, the physical memory currently used by SQL Server) is either small or under constant pressure to shrink. Note, the SQL Server process monitors physical memory pressure on the server box and frees up its allocated memory by shrinking the buffer pool.

The physical memory pressure can also come from within the SQL Server process itself. For example, when you change memory settings (for example, max server memory configuration parameter) of SQL Server instance or change memory distribution of internal components (for example, high percentage of reserved and stolen pages from buffer pool), it can lead to internal memory pressures. Memory for internal components is allocated using two types of allocations.

Single-page allocations

These represent memory allocated by using the single-page allocator. The single-page allocator steals pages directly from the buffer pool. For example, procedure cache allocates memory by stealing pages from the buffer pool.

Multipage allocation

This represents memory allocated by using the multipage allocator. This memory is allocated outside the buffer pool.

Virtual memory pressure

SQL Server process can experience virtual memory pressure from external components or from within itself. It experiences external virtual memory pressure as the virtual address available on the server box gets closer to its limit, the server box is considered to be under virtual memory pressure. For example, if other processes running on the server box use up most of the memory (including the swap file) available, then the SQL Server process may fail to start or will not be able to commit more memory to keep up with the demand of the workload. This condition will likely lead to a very sluggish response in the whole system.

The virtual memory pressure can even come from within the SQL Server process. One example might be running low on VAS because of fragmentation (a lot of VAS is available but in small blocks) and/or consumption (direct allocations, DLLs loaded in SQL Server VAS, high number of threads). SQL Server detects this condition and may release reserved regions of VAS, and may start shrinking caches.

The following sections describe how to detect and troubleshoot memory pressure.

Detection of physical memory pressure

The simplest way to identify physical memory pressure is by opening the Task Manager in Performance view and total value and available value in the Physical Memory section. Total represents the actual physical memory (that is, RAM) available on your server box while available represents the memory available to new process. A low available value can indicate physical memory pressure. The exact value depends on many factors; however you can start looking into this when the value drops below 50 to 100 MB. External memory pressure is clearly present when this amount is less than 10 MB.

Several System Monitor counters are available to identify physical memory pressure. One common question is what is the typical value of these counters. As before, there is no simple answer for all counters, and you will need to rely on the baseline of your workload to see any significant deviations. You can use the following System Monitor counters to identify memory pressure.

- Memory: Available Bytes This represents the amount of physical memory, in bytes, available to processes running on the computer. It is calculated by adding the amount of space on the Zeroed, Free, and Standby memory lists. Free memory is ready for use; Zeroed memory consists of pages of memory filled with zeros to prevent subsequent processes from seeing data used by a previous process; Standby memory is memory that has been removed from a process' working set (its physical memory) on route to disk, but is still available to be recalled. This counter displays the last observed value only; it is not an average.
- SQLServer:Buffer Manager: Buffer Cache Hit Ratio This represents the percentage of pages that were found in the buffer pool without having to incur a read from disk. For most production workloads this value should be in the high 90s.
- SQLServer:Buffer Manager: Page Life Expectancy This represents the number of seconds a page will stay in the buffer pool without references. A lower value indicates that the buffer pool is under memory pressure.
- SQLServer:Buffer Manager: Checkpoint Pages/Sec This represents number of pages flushed by checkpoint or other operations that require all dirty pages to be flushed. This indicates increased buffer pool activity of your workload.
- SQLServer:Buffer Manager: Lazywrites/Sec This represents number of buffers written by buffer manager's lazy writer and has a similar implication as described before for checkpoint pages/sec.

Normally, the buffer pool accounts for the most of the memory committed by SQL Server. To determine the amount of memory that belongs to the buffer pool, we can take a look at the DBCC MEMORYSTATUS output. In this output, find the buffer counts section and look at the target and committed values. The following shows one such part of the DBCC MEMORYSTATUS output after the SQL Server has reached its normal load.

Buffer Counts	Buffers
Committed	201120
Target	201120
Hashed	166517
Stolen Potential	143388
External Reservation	0
Min Free	256
Visible	201120
Available Paging File	46040

- Committed This value shows the total buffers that are committed. Buffers that are committed have

physical memory associated with them. The Committed value is the current size of the buffer pool. This value includes the physical memory that is allocated if AWE support is enabled.

- Target This value shows the target size of the buffer pool. It is computed periodically by SQL Server as the number of 8-KB pages it can commit without causing paging. SQL Server lowers its value in response to memory low notification from the Windows operating system. A decrease in the number of target pages on a normally loaded server may indicate response to an external physical memory pressure.

The physical memory pressure can also be from within SQL Server itself. For example, if the procedure cache grows very big, it can put pressure on the buffer pool. Since the internal memory pressure is set by SQL Server itself, a logical step is to look at the memory distribution inside SQL Server by checking for any anomalies in buffer distribution. You can look at the stolen pages count from DBCC MEMORYSTATUS output.

Buffer Distribution	Buffers
Stolen	32871
Free	17845
Cached	1319
Database (clean)	148864
Database (dirty)	6033
I/O	0
Latched	0

A high percentage (greater than 75 to 80 percent) of stolen pages of the total committed pages is a sign of internal memory pressure.

Similarly, if internal components allocate a large amount of memory outside of the buffer pool, it again puts pressure on the buffer pool. You can compute it by subtracting the current size of the buffer pool from the System Monitor counter Process: Private Bytes. A high value here can indicate internal memory pressure. Typically, the components that are loaded into the SQL Server process, such as COM objects, linked servers, extended stored procedures, SQLCLR, and others contribute to memory consumption outside of the buffer pool. There is no easy way to track memory consumed by these components especially if they do not use SQL Server memory interfaces.

Detection of Virtual Memory Pressure

The simplest way to identify virtual memory pressure is by checking if the swap page file(s) have enough space to accommodate current memory allocations. To check this, open Task Manager in Performance view and check the Commit Charge section. If Total is close to the Limit, then there exists the potential that page file space may be running low. Limit indicates the maximum amount of memory that can be committed without extending page file space. Note that the Commit Charge Total in Task Manager indicates the potential for page file use, not the actual use as part of the committed memory is likely to be in the RAM. Actual use of the page file will increase under physical memory pressure. You can also look at the following System Monitor counters available to identify virtual memory pressure.

- Paging File:%Usage Represents the amount of the Page File instance in use in percent.
- Memory: Commit Limit Represents the amount of virtual memory that can be committed without having to extend the paging file(s). It is measured in bytes. Committed memory is the physical memory that has space reserved on the disk paging files. This counter displays the last observed value only; it is not an average.

Isolation and Troubleshooting of Memory Pressure

Once you have identified that your SQL Server instance is experiencing memory pressure, some common steps that you can take include: reduce the external memory pressure, add more physical memory, and enable AWE mode on 32-bit installations.

If you are experiencing external physical memory pressure, you will need to identify major consumers of the physical memory on the system. To do this, look at Process:Working Set performance counters or the MemUsage column on the Processes tab of Task Manager and identify the largest consumers. The total use of physical memory on the system can be roughly accounted for by summing the following counters.

- Process: Working Set counter for each process
- Memory: Cache Bytes counter for system working set
- Memory: Pool Nonpaged Bytes counter for size of unpaged pool
- Memory: Available Bytes (equivalent of the Available value in Task Manager)

If there's no external physical memory pressure, the Process: Private Bytes counter or the VM Size in Task Manager should be close to the size Process: Working Set, which means that we have no memory paged out. You will need to analyze processes other than SQL Server that have large working sets. One solution may be to not run such applications on the server box.

The AWE mechanism allows a 32-bit application to manipulate physical memory beyond the inherent 32-bit address limit. The AWE mechanism is not needed on a 64-bit platform to access more physical memory. It is, however, available but unlike a 32-bit platform, it does not require setting the AWE Enabled configuration option. Memory pages that are allocated through the AWE mechanism are referred to as locked pages on the 64-bit platform. On both 32- and 64-bit platforms, memory that is allocated through the AWE mechanism cannot be paged out. This can be beneficial to the application. (This is one of the reasons for using AWE mechanism on 64-bit platform.) This also affects the amount of RAM that is available to the system and to other applications, which may have detrimental effects. For this reason, in order to use AWE, the Lock Pages in Memory privilege must be enabled for the account that runs SQL Server. From a troubleshooting perspective, an important point is that the SQL Server buffer pool uses AWE-mapped memory; however, only database (hashed) pages can take full advantage of memory allocated through AWE. Memory allocated through the AWE mechanism is not reported by Task Manager or in the Process: Private Bytes performance counter. However, both the DMVs and the DBCC Memory Status command are AWE aware. For example, you can use the following DMV query to find the total amount of memory consumed (including AWE) by the buffer pool:

Code View:

```
SELECT
    SUM(multi_pages_kb + virtual_memory_committed_kb + shared_memory_committed_kb
        + awe_allocated_kb) AS [Used by BPool, Kb]
FROM sys.dm_os_memory_clerks
WHERE type = 'MEMORYCLERK_SQLBUFFERPOOL'
```

Here is the sample output:

```
Used by BPool, Kb
-----
8269684
(1 row(s) affected)
```

If you have determined (using DBCC MEMORYSTATUS) that there is an internal memory pressure because internal components have stolen most of the pages from buffer pool, you can identify internal components that are stealing the most pages from buffer pool using the following DMV query:

```
SELECT TOP 10 type,
       SUM(single_pages_kb) AS stolen_mem_kb
  FROM sys.dm_os_memory_clerks
 GROUP BY type
 ORDER BY SUM(single_pages_kb) DESC
```

You will see output like the following:

Type	stolen_mem_kb
CACHESTORE_PHDR	20020
MEMORYCLERK_SOSNODE	1104
CACHESTORE_SYSTEMROWSET	1040
CACHESTORE_SQLCP	880
MEMORYCLERK_SQLGENERAL	832
MEMORYCLERK_SQLSTOREENG	816
USERSTORE_SCHEMAMGR	400
CACHESTORE_BROKERETBLACS	368
OBJECTSTORE_LOCK_MANAGER	352
USERSTORE_DBMETADATA	288
OBJECTSTORE_SERVICE_BROKER	256

You do not have control of memory used by internal components. However, determining the internal components that are using the most memory will help narrow down the investigation of the problem. You can identify the internal components that have allocated memory outside of the buffer pool by using the multipage allocator with the following query:

```
SELECT type, SUM(multi_pages_kb) AS memory_allocated_KB
  FROM sys.dm_os_memory_clerks
 WHERE multi_pages_kb != 0
 GROUP BY type
```

Your output will look like the following:

type	memory_allocated_KB
MEMORYCLERK_SQLSTOREENG	56
OBJECTSTORE_SNI_PACKET	96
MEMORYCLERK_SQLOPTIMIZER	72
MEMORYCLERK_SQLGENERAL	1696
MEMORYCLERK_SQLBUFFERPOOL	256
MEMORYCLERK_SOSNODE	8352
CACHESTORE_STACKFRAMES	16
MEMORYCLERK_SQLSERVICEBROKER	192
MEMORYCLERK_SNI	32
CACHESTORE_XPROC	50120

Here you can see that XPROC are consuming 50 MB of memory outside of the buffer pool. If a significant amount of memory is allocated through the multipage allocator (100â€“200 MB or more), further

investigation is warranted.

For virtual memory pressure, some general recommendations are (1) to increase the size of your page file (s), (2) use /3 GB option if applicable, (3) if running on 32-bit hardware, avoid using AWE because it takes a large chunk of VAS but on the other hand you may need AWE to minimize physical memory pressure, and (4) switch to 64-bit hardware that offers you a user-mode address space of 8 terabytes (7 terabytes on Itanium-based systems).

There are many more DMVs that you can use to get memory-related information. One notable one is sys.dm_os_ring_buffer. For details, you can take a look at the whitepaper Troubleshooting Performance Problems in SQL Server 2005, which is available on the companion Web site and also at <http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx>.

I/O Bottlenecks

The performance of SQL Server heavily depends on the performance of its I/O subsystem. Unless your database fits entirely into the physical memory or your application accesses and manipulates only a subset of data that fits into memory, the SQL Server needs to bring pages in and out of memory all the time, which can generate significant I/O traffic. I/O is not limited to only data pages. Depending upon the DDL/DML activity, SQL Server can generate a large numbers of log records. Leaving aside some logging optimizations, almost all changes to a database are logged and these log records need to be flushed to the disk before SQL Server can declare a transaction committed. Similarly, SQL Server uses tempdb to store intermediate results of query processing or DDL operations. For example, it uses tempdb to sort the data both when the Order By clause is specified or when creating/rebuilding indexes and to store row versions (SQL Server 2005 only) created to support SNAPSHOT ISOLATION, Online Index build, Multiple Active Row Sets (MARS), and triggers. Besides, applications use tempdb to store intermediate results. So it is critical that your I/O subsystem is configured to support the demands of your application workload. Your application will experience performance degradation if the I/O subsystem cannot keep up with the fluctuations or changes in the workload. An I/O bottleneck does not necessarily imply that you need to add more I/O bandwidth. As we will see later in this section, the I/O bottleneck may be a symptom of problems in other parts of SQL Server.

Detection of I/O bottlenecks

System Monitor counters provide a very useful set to identify the I/O problems. You can use these counters to see the overall performance of your I/O subsystem or individual physical disks. Some of the commonly used counters to identify I/O bottleneck are as follows. Note that these numbers are average and can hide any spike in values inside polling interval so it is advised that you look at many of these counters for validation.

- PhysicalDisk Object: Avg. Disk Queue Length represents the average number of physical Read and Write requests that were queued on the selected physical disk during the sampling period. If your I/O system is overloaded, more Read/Write operations will be waiting. If your disk-queue length frequently exceeds a value of two per physical disk during peak usage of SQL Server, then you might have an I/O bottleneck.
- PhysicalDisk Object: Avg. Disk Sec/Read or Avg. Disk Sec/Write is the average time, in seconds, of a read or write of data from/to the disk. Some general guidelines follow:
 - ◆ Less than 10 ms is very good
 - ◆ Between 10 and 20 ms is okay
 - ◆ Between 20 and 50 ms is slow, needs attention
 - ◆ Greater than 50 ms is considered a serious I/O bottleneck
- PhysicalDisk: Disk Reads/Sec or Disk Writes/Sec is the rate of read or write operations on the disk. You need to make sure that this number is less than 85 percent of the disk capacity. The disk access time increases exponentially beyond 85 percent capacity.

System Monitor counters provide you with the I/O performance information at the physical disk level but not at the file level. The information at file level is useful if you have a mix of log, data, and tempdb files on the same disk. In this case, for example, it will allow you to isolate it if it is the logging or the tempdb that is causing the I/O bottleneck. Similarly, it will be useful if you have files from different file groups share the same physical disk and you want to know if object(s) mapped to a particular file are the reason of an I/O bottleneck. Using this information, you can choose to remap your objects to different physical disks to minimize I/O latency.

Here is the DMV query that can be used to identify such file(s). The two columns, io_stall_read_ms and io_stall_write_ms, represent the time SQL Server waited for Reads and Writes issued on the file since the start of SQL Server. To get meaningful data, you will need to snapshot these numbers for small duration and then compare it to these numbers with baseline numbers. If you see that there are significant differences, it will need to be analyzed.

```
SELECT
    database_id,
    file_id,
    io_stall_read_ms,
    io_stall_write_ms
FROM sys.dm_io_virtual_file_stats(NULL, NULL)
```

You can also identify overall I/O bottlenecks by examining the latch waits. These latch waits account for the physical I/O waits when a page is accessed for reading or writing and the page is not available in the buffer pool. When the page is not found in the buffer pool, an asynchronous I/O is posted and then the status of the I/O is checked. If I/O has already completed, the worker proceeds normally. Otherwise, it waits on PAGEIOLATCH_EX or PAGEIOLATCH_SH, depending upon the type of request. The following DMV query can be used to find I/O latch wait statistics.

```
SELECT
    wait_type,
    waiting_tasks_count,
    wait_time_ms,
    signal_wait_time_ms
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'PAGEIOLATCH%'
ORDER BY wait_type
```

The interesting latch waits are PAGEIOLATCH_SH and PAGEIOLATCH_EX. These waits occur when a task is waiting on a latch for a buffer that is in an I/O request. The long waits of this type indicate a problem with the disk subsystem. The column wait_time_ms includes the time a worker spends in SUSPENDED state and in RUNNABLE state while the column signal_wait_time_ms represents the time a worker spends in RUNNABLE state. So the difference of the two (wait_time_ms - signal_wait_time_ms), actually represents the time spent waiting for I/O to complete. One thing to keep in mind is that the above DMV returns waits since the SQL Server was started. So to get meaningful data, you will need to look at the delta between the periods of interest.

Isolation and troubleshooting of I/O bottlenecks

When you are experiencing an I/O bottleneck, it is tempting to assume that it is because of the disk I/O subsystem. In fact, it may very well be, but before you reach that conclusion, you need to consider two factors that can eventually lead to I/O bottleneck.

First, you need to check if SQL Server memory has been properly configured for the workload. If there is not enough physical memory, the pages in the buffer pool will need to be recycled aggressively which leads to physical I/Os and ultimately to an I/O bottleneck. Please refer the previous section on how to detect and troubleshoot memory bottleneck. If you find that your SQL Server has not been properly configured with physical memory or it is experiencing physical memory pressure caused by other applications that share the box with SQL Server, you will need to address this issue first and verify if this addresses the I/O bottleneck.

Second, you need to look at queries/batches that are causing the most I/Os. You will need to take a look at their query plans to see if the large number of I/Os is a result of a poor query plan or a missing index. The following DMV query returns the top-10 queries/batches that are generating the most I/Os. You can also use a variation of this query to find the top-10 queries that do the most I/Os per execution.

```
SELECT TOP 10
    (total_logical_reads/execution_count) AS avg_logical_reads,
    (total_logical_writes/execution_count) AS avg_logical_writes,
    (total_physical_reads/execution_count) AS avg_phys_reads,
    execution_count,
    (SELECT SUBSTRING(text, statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(MAX),text)) * 2
            ELSE statement_end_offset
            END - statement_start_offset)/2)
        FROM sys.dm_exec_sql_text(sql_handle)) AS query_text,
    plan_handle
FROM sys.dm_exec_query_stats
ORDER BY (total_logical_reads + total_logical_writes) DESC
```

The above query outputs both the SQL text and the corresponding query plan. As indicated earlier, you will need to evaluate the query plan and the SQL statement(s) to see if the number of I/Os can be reduced. Here is the simplified query output of the above DMV. We can make a few interesting observations about the workload. First, it shows that the query with the JOIN operator is generating the maximum I/Os. Second, it shows that the aggregate query did not encounter any physical I/Os. It means that all pages referenced by this query were already in the buffer pool. So it is likely that this query was run after the JOIN query which brought all 6,008 pages for table t1 into the buffer pool. Third, the average number of physical I/Os is 1.256, which is much less than 16,536, the average number of logical I/Os. It means that subset pages from tables t1 and t2 were already in the buffer pool and/or SQL Server reduced physical I/Os incurred by prefetching some of the pages needed. Fourth, it provides you the plan handle, which can be used to generate the actual query plan and to understand the reasons for a large number of I/Os.

avg_logical_reads	avg_logical_writes	avg_phys_reads
16536	4	1256
6008	0	0
Execution_count	query_text	
3	<pre>select c1, c5 from t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4 order by c2</pre>	
1	<pre>select sum(c1) from t1</pre>	
Plan_handle		
0x06000500E401FC06B8E1A00300000000000000000000000000000000000		
0x060005005B9B8F16B841B70300		

Among many things that can be done to improve the query performance by reducing its number of I/Os, one of the first things that you should check is if a useful index is missing. With SQL Server 2005, a set of DMVs is provided to identify missing indexes including their potential usefulness (that is, how many times a missing index would have been used). Let us consider an example to show how DMVs can be used to identify missing indexes.

Assume we have a table `t_sample` that has five columns. We first load 1,000 rows into this table followed by creating two indexes: one clustered index on column `c1` and a nonclustered index on column `c4`. Here is the script:

```

CREATE TABLE t_sample (c1 int, c2 int, c3 int, c4 int, c5 char(5000))
GO
-- insert the data and create the indexes
DECLARE @i int
SELECT @i = 0
WHILE (@i < 1000)
BEGIN
    INSERT INTO t_sample
        VALUES (@i, @i + 1000, @i+2000, @i+3000, 'hello')
    SET @i = @i + 1
END
-- create the indexes
CREATE CLUSTERED INDEX t_sample_ci ON t_sample(c1)
CREATE NONCLUSTERED INDEX t_sample_nci_c4 ON t_sample(c4)

```

Now, we run multiple selects in a loop as follows:

```

DECLARE @i int
SELECT @i = 0
WHILE (@i < 100)
BEGIN
    SELECT SUM(c1 + c2 + c3) FROM t_sample
    WHERE c1 BETWEEN @i AND @i+50
    SELECT SUM(c2) FROM t_sample
    WHERE c2 BETWEEN @i+1000 AND @i + 1100
    SELECT SUM(c3) FROM t_sample
    WHERE c3 BETWEEN @i +2000 AND @i+2400
    SET @i = @i + 1
END

```

You will notice that there is no index on column `c2` and `c3`, so for the second and third SELECT statement, SQL Server will need to do a table scan. The indexes on column `c2` and column `c3` are in fact the missing indexes. Here is a DMV query that you can execute to identify the missing indexes and their usefulness:

```

-- Use DMVs to determine what indexes are missing and could be useful
SELECT t1.object_id, t2.user_seeks, t2.user_scans,
       t1.equality_columns, t1.inequality_columns

FROM sys.dm_db_missing_index_details AS t1,
     sys.dm_db_missing_index_group_stats AS t2,
     sys.dm_db_missing_index_groups AS t3
WHERE database_id = DB_ID()
      AND object_id = OBJECT_ID('t_sample')
      AND t1.index_handle = t3.index_handle
      AND t2.group_handle = t3.index_group_handle

```

And here is the output that shows that if we had indexes on columns c2 and c3; each would have been useful in 100 seeks through index. The missing index-related DMVs can provide much more useful information. Please refer to BOL for the details.

Code View:

object_id	user_seeks	user_scans	equality_columns	inequality_columns
1989582126	100	0	NULL	[c2]
1989582126	100	0	NULL	[c3]

Once you have addressed the issue related to memory and I/Os done by various queries and you still are encountering an I/O bottleneck, then you need to look at upgrading your I/O subsystem to meet the demands of your workload.

When you upgrade your I/O subsystem, you should first check the capacity of your I/O subsystem. If your I/O subsystem is not configured properly, you may not be getting the I/O bandwidth that it can potentially deliver. You can use SQLIO.exe tool, provided by Microsoft, to determine the I/O capacity of a given I/O configuration. You can download this tool at:

<http://www.microsoft.com/downloads/details.aspx?familyid=9a8b005b-84e4-4f24-8d65-cb53442d9e19&displaylang=en>

Bottlenecks

The tempdb database is a shared resource for the SQL Server instance. Applications use tempdb to store intermediate or transient data in temporary tables or table variables for subsequent processing. For example, you may want to materialize the result of a complex query into a temporary table if you need to iterate over that result multiple times. This can save you the cost of executing the complex query multiple times. You may wonder why an application can't store the intermediate result set into a table in the regular database. The simple answer is that it can, but there are the following issues with it:

- It requires the user to have permission to create the table in the database. Very few users have CREATE TABLE permission.
- There is more logging overhead in user database for two reasons. First, the recovery model of user database may be set to FULL. Second, the log records in the user database must have both the undo and redo information. Starting with SQL Server 2005, the redo information is not logged in tempdb.
- It can clutter the user database if these intermediate tables are not dropped. In fact, they may lead to query failure if the intermediate table it is trying to create already exists.
- The intermediate tables may overload the I/O path for user database.

The temporary tables, explicitly named as #<table> and ##<table>, are created in tempdb. Both of these objects are session-scoped, but the ##<table> lives until all sessions that are using it expire or terminate, while the #<table> is destroyed when the scope (for example, stored procedure or session) it was created in expires or terminates. Because of this, the #<table> and ##<table> are referred to as local temporary table and global temporary tables, respectively. Finally, users don't require CREATE TABLE permission in tempdb. For these reasons, most applications use temporary tables to store intermediate results. Besides temporary tables, you can also use table variable to store intermediate results. A table variable is like a local temporary table and is persisted in tempdb, but its scope is limited to the batch/request it is defined in and has lower overhead for locking and logging. It has some limitations. For example, you cannot create indexes on a table variable, and statistics are not maintained. It is typically used to represent a temporary rowset to be returned as a rowset of a table-valued function. Like user tables, the metadata information is created for temporary tables and table variables. For more details on its usage, please refer to SQL Server 2005 BOL. Since these objects are created explicitly by users, they are categorized as User Objects.

SQL Server also uses tempdb to store intermediate rowsets during the processing of a SQL statement. For example,

- When doing an external sort to process an order by.
- When creating or rebuilding an index.
- When doing hash join if the hash table does not fit in-memory.
- When storing a cursor result set, spooling the data, or to store LOB variables.
- And many more.

SQL Server stores this data, not in #table or ## tables, but in many flavors of internal objects like work tables, work files, and Sort files. The internal objects differ from user objects in multiple ways. First, these objects are not represented in a catalog, so there is no DDL overhead. Second, these objects are scoped to the Transact-SQL statement and are destroyed after the statement is completed. Third, operations on these objects are not logged with the exception of sort files where allocations are logged. So these objects have much lower overhead than user objects.

Starting with SQL Server 2005, there is a new kind of entity called version store in tempdb. The version store is used to store older but transactionally consistent data/index rows. These row versions are used to implement:

- Triggers to generate deleted/inserted rowsets. Before SQL Server 2005, the inserted/deleted rowsets were generated by traversing the log chain backwards. The main drawback was that it randomized the access to the log, which could potentially impact the log throughput.
- SNAPSHOT ISOLATION and READ COMMITTED SNAPSHOT ISOLATION
- Online Index Build
- Multiple Active Row Sets (MARS)

A row version can only be removed when it is not needed. A background thread runs every minute to clean up the row versions. Long-running transactions that depend on row versions can prevent version store cleanup, and that can lead to space issues in tempdb. For details on objects and version store in tempdb, you can take a look at the whitepaper Working with tempdb in SQL Server 2005, which is available on the companion Web site and also at <http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx>.

If you have queries/batches that make heavy use of tempdb or have a tempdb that is not properly configured for the workload, your application is likely to run into tempdb-related performance issues. Some of the common performance issues related to tempdb are:

- I/O bottleneck Both internal and user objects in tempdb are created, populated, and then accessed for DML operations in the statement or session scope except for global temporary tables. So the chances are that the pages for these objects are in the buffer pool. However, depending on the total number of pages allocated to these objects and the size of the buffer pool, there can be significant physical I/O on tempdb. For example, when creating an index on a large table, the Sort operation in tempdb may need to be done in multiple passes (i.e., external sort), which can generate lot of physical I/Os for Write/Read operations. Besides data and index pages, DML operations on user objects generate log records in tempdb, which can be another source of physical I/Os. And finally, there is a version store that can cause physical I/Os, especially if there are long version chains, typically an indication of a long-running transaction.
- Allocation bottleneck Data modification operations that manipulate data in tempdb often lead to the allocation and deallocation of pages. Excessive page allocation/deallocation activity in tempdb can lead to contention in the allocation structures that track allocation information.
- DDL bottleneck Contention for DDL operations only occurs for user objects in tempdb. Metadata for internal objects, such as work tables, is cached and is not created in the system catalog. However, metadata for user objects, such as local and global temporary tables and table variables, is created in the system catalog. If your workload or application creates and drops a large number of user objects in

tempdb, contention in system catalog tables may impact the performance of your workload.

- tempdb growth Every time SQL Server starts, it re-creates the tempdb. Since all objects in tempdb are temporary by definition, no recovery or restore is done. When the tempdb is re-created, its size is set to the last configured size or the default size when the tempdb was first created. If your workload needs a much larger tempdb in steady state, you will have two issues. First, your application may need to pause while the tempdb is growing. Second, frequent auto-grow can lead to physical fragmentation of files, especially if the physical disk is shared with other databases and/or applications. Physical fragmentation implies that a file consists of multiple noncontiguous fragments on physical disk, which leads to higher latency in accessing the disk. Similarly, if your workload needs a much larger size for tempdb log files than its last configured value, your application will need to pause while the log file is growing. It can also lead to physical fragmentation just like in the case of data files.

Now that we understand the type of performance issues related to tempdb, the next sections describe how to detect and troubleshoot these problems.

Detection of performance issues

Previously, in the I/O Bottleneck section, we described how you can identify an I/O bottleneck at a physical-disk level using System Monitor counters or at file level using the sys.dm_io_virtual_file_stats DMV. If your tempdb is created on its own physical disk, you can detect a tempdb specific I/O bottleneck using the System Monitor counters; otherwise you will need to use the DMVs described previously in I/O Bottlenecks section. A general guideline is to create tempdb in its own group of physical disks for the following reasons.

- If tempdb is heavily used, an I/O bottleneck in tempdb can impact the performance of all applications accessing databases that share physical disks with tempdb.
- Since tempdb does not need crash recovery, there is no requirement for its log records to be flushed and persisted on physical media when a transaction is committed. This gives you a lot more choices about what kinds of disks you can use for tempdb. For example, you can use RAM disk or a disk with large cache. For more information, please refer to KB Article 917047.

You can detect tempdb automatic growth both for data and log files using SQL Profiler and Trace, DATA FILE AUTO GROW, and LOG FILE AUTO GROW events. You can also periodically poll the sys.database_files catalog view, which returns the current size of both data files and log files in tempdb. If the size of these files is changing over time, it implies the file grew either because of auto-grow or because of explicit change in the size of the file(s) using the ALTER DATABASE command. However, irrespective of what causes a file to grow, it can lead to physical fragmentation, which should be avoided if possible.

Allocation bottlenecks are caused by multiple threads waiting to acquire latches on allocation structures as described below. Allocation structures are discussed in more detail in Inside SQL Server 2005: The Storage Engine.

- GAM pages track extent allocation information. Each extent is represented by a bit. If the value of bit is 1, it means the extent is free for allocation.
- SGAM pages track allocation information of mixed extents. A mixed extent contains pages from multiple objects.
- PFS pages track the page allocation status within an extent and also the free space available on the page. When a page is to be allocated or deallocated, SQL Server updates the corresponding PFS information to indicate the allocation status of the pages. Similarly, when a row is inserted or deleted, SQL Server may need to update the corresponding PFS information.

As you can see, if your workload is doing excessive allocation/deallocation of pages and data modification operations that cause free space in pages to vary, SQL Server may encounter an allocation bottleneck. You can use the following DMV query to identify if you have an allocation bottleneck. This query finds out if one

or more threads are waiting to acquire latch on pages in tempdb. Note, this DMV shows the current workers that are waiting. You will need to poll this DMV often to identify allocation bottleneck.

```
SELECT session_id,
       wait_duration_ms,
       resource_description
  FROM sys.dm_os_waiting_tasks
 WHERE wait_type LIKE 'PAGE%LATCH_%' AND
       resource_description like '2:%'
```

Since the database id of tempdb is 2, the search argument 2.% represents any page in tempdb across any file; the trick is to know if the page is GAM, SGAM, or PFS. A PFS page is the first page after the file header page in a data file (page 1). This is followed by a GAM page (page 2), and then an SGAM page (page 3). There is a PFS page approximately 8,000 pages in size after the first PFS page. There is another GAM page 64,000 extents after the first GAM page, and another SGAM page 64,000 extents after the first SGAM page.

You can also monitor the following System Monitor counters for any unusual increase in the allocation/deallocation activity of user and internal objects in tempdb.

- SQLServer:Access Methods: Worktables Created/Sec The number of work tables created per second. Work tables are temporary objects and are used to store results for query spool, LOB variables, and cursors. Typically, this number is less than 200, but again, you will need to compare with your baseline.
- SQLServer:Access Methods: Workfiles Created/Sec The number of work files created per second. Work files are similar to work tables but are created strictly by hashing operations. Work files are used to store temporary results for hash joins and hash aggregated.
- SQLServer:Access Methods: Worktables from Cache Ratio The percentage of work tables that were created where the initial two pages of the work table were not allocated but were immediately available from the work table cache. In SQL Server 2000, there is no caching of temporary tables.
- SQLServer:General Statistics: Temp Tables Creation Rate The number of temporary table or variables created/sec.
- SQLServer:General Statistics: Temp Tables for Destruction The number of temporary tables or variables waiting to be destroyed by cleanup system thread.

You can detect a DDL bottleneck similarly to the way you identify allocation contention. The following DMV query can help you to identify DDL bottlenecks. This query finds out if one or more threads are waiting to acquire latch-on pages in tempdb. Note, this DMV shows the current workers that are waiting. You will need to poll this DMV often to identify a DDL bottleneck.

```
SELECT session_id,
       wait_duration_ms,
       resource_description
  FROM sys.dm_os_waiting_tasks
 WHERE wait_type LIKE 'PAGE%LATCH_%' AND
       resource_description like '2:%'
```

This query is same as the one we used for detecting allocation bottlenecks. The only difference is that if the page that is incurring a latch contention is a page that belongs to the system catalog; it means you have DDL contention.

Isolation and troubleshooting of bottlenecks

All recommendations that we discussed under I/O bottlenecks in the earlier section will apply here as well, including the recommendations regarding detecting and relieving memory pressure. But specifically for tempdb, you will need to identify what queries/batches are generating the most I/Os in tempdb and then see if you can either change the Transact-SQL or the query plan to minimize it. You can use the following DMV query to identify the currently executing query that is causing the most allocations and deallocations in tempdb.

Code View:

```
SELECT TOP 10
    t1.session_id,
    t1.request_id,
    t1.task_alloc,
    t1.task_dealloc,
    t2.plan_handle,
    (SELECT SUBSTRING (text, t2.statement_start_offset/2 + 1,
        (CASE WHEN statement_end_offset = -1
            THEN LEN(CONVERT(nvarchar(MAX),text)) * 2
            ELSE statement_end_offset
        END - t2.statement_start_offset)/2)
    FROM sys.dm_exec_sql_text(sql_handle)) AS query_text
FROM (SELECT session_id, request_id,
    SUM(internal_objects_alloc_page_count +
        user_objects_alloc_page_count) AS task_alloc,
    SUM(internal_objects_dealloc_page_count +
        user_objects_dealloc_page_count) AS task_dealloc
    FROM sys.dm_db_task_space_usage
    GROUP BY session_id, request_id) AS t1,
    sys.dm_exec_requests AS t2
WHERE t1.session_id = t2.session_id AND
    (t1.request_id = t2.request_id) AND t1.session_id > 50
ORDER BY t1.task_alloc DESC
```

This query gives you top-10 queries that are causing the most page allocations in tempdb. You will need to analyze the query plan of each query and see if you can either change the Transact-SQL or the query plan or create missing indexes to minimize I/O in tempdb. Here is one example that shows how the preceding DMV query can be used to identify a query that is generating the most allocations and deallocations in the tempdb.

```
CREATE TABLE t1 (c1 int PRIMARY KEY, c2 int, c3 char(8000))
GO
CREATE TABLE t2 (c4 int, c5 char(8000))
GO
-- load large number of rows into each of the tables
DECLARE @i int
SELECT @i = 0
WHILE (@i < 6000)
BEGIN
    INSERT INTO t1 VALUES (@i, @i + 1000, 'hello')
    INSERT INTO t2 VALUES (@i, 'there')
    SET @i = @i + 1
END
-- Run a query with hash-join in a separate session
SELECT c1, c5
FROM t1 INNER HASH JOIN t2 ON t1.c1 = t2.c4
ORDER BY c2
```


see if you can minimize those DDL operations. For example, if you are creating a temporary table inside a loop, see if you can move it out of the loop. SQL Server 2005 caches temporary objects, when possible, so that dropping and creating temporary objects is very fast. When SQL Server drops a temporary object, it does not remove the catalog entry for the object, and when it is used next time, there is no need to re-create the temporary object. SQL Server caches 1 data page and 1 IAM page for the temporary object and deallocates the rest of the pages. If the size of the temporary object is larger than 8 MB, the deallocation is done asynchronously in the background.

SQL2005 caches temporary objects when the following conditions are satisfied:

- Named constraints are not created.
- Data Definition Language (DDL) statements that affect the temporary table are not run after the table has been created, such as the CREATE INDEX or CREATE STATISTICS statements.
- The temporary object is not created by using dynamic SQL, such as sp_executesql N'CREATE TABLE #t(a int)'.
- The temporary object is created inside another objectâ€”such as a stored procedure, trigger, or user-defined functionâ€”or is the return table of a user-defined, table-valued function.

You can use the following script to identify if temporary objects are being cached (or not) in a stored procedure. It shows number of temporary objects created when you invoke a particular stored procedure 10 times.

Code View:

```

DECLARE @table_counter_before_test bigint
SELECT @table_counter_before_test = cntr_value
FROM sys.dm_os_performance_counters
WHERE counter_name = 'Temp Tables Creation Rate'

DECLARE @i int
SELECT @i = 0
WHILE (@i < 10)
BEGIN
    <execute your stored procedure>
    SELECT @i = @i+1
END
DECLARE @table_counter_after_test bigint;
SELECT @table_counter_after_test = cntr_value
FROM sys.dm_os_performance_counters
WHERE counter_name = 'Temp Tables Creation Rate'
PRINT 'Temp tables created during the test: ' +
      CONVERT(varchar(100), @table_counter_after_test - @table_counter_before_test)

```

Consider the following example. The stored procedure test_tempetable_caching creates a #<table> inside it. We want to check if this temporary table is cached across multiple invocations using the script described previously.

```

CREATE PROCEDURE test_tempetable_caching
AS
CREATE TABLE #t1 (c1 int, c2 int, c3 char(5000))
- CREATE UNIQUE CLUSTERED INDEX ci_t1 ON #t1(c1);
DECLARE @i int
SELECT @i = 0
WHILE (@i < 10)
BEGIN
    INSERT INTO #t1 VALUES (@i, @i + 1000, 'hello')
    SELECT @i = @i+1

```

```

END
PRINT 'done with the stored proc'
GO

```

Here is the output:

Code View:

```

done with the stored proc
...
done with the stored proc
done with the stored proc
Temp tables created during the test: 1

```

Note, if we create the `on` inside stored procedure, the output shows that temporary object was created and dropped 10 times (i.e., not cached). If this stored procedure is executed many times, it can potentially cause significant DDL load in .

`done with the stored proc`

```

...
done with the stored proc
done with the stored proc
Temp tables created during the test: 10

```

If you really have to create an index on a temporary table, you can use the following DDL and still keep the temporary table cached. Note that the unique constraint creates the clustered index implicitly.

```
CREATE TABLE #t1 (c1 int UNIQUE, c2 int, c3 char(5000))
```

Auto and explicit growth of

You will need to know what the maximum size of your tempdb in your workload is and then preallocate your tempdb to that size. It gives you two main benefits, as outlined earlier. First, you minimize physical fragmentation of your files. Second, user transactions don't get stalled while waiting for tempdb to grow. It is always a good idea to keep auto-grow enabled and empty space on the disk subsystem for tempdb to handle exceptional situations. One improvement in SQL Server 2005 running on Windows XP and Windows 2003 is that you can skip zeroing out the files(s), called instant file initialization, when creating a database or when growing one or more files. Instant file initialization reclaims used disk space without writing to that space with zeros. The disk contents are overwritten as new data is written to the files.

Blocking

SQL Server supports concurrent access by thousands of users accessing shared resources like user tables, system tables, and internal data structures, like buffers, query plans, and so on. To guarantee the integrity of the data and of its internal structures, the SQL Server deploys various synchronization mechanisms like logical locks, spinlocks, and latches to control access to these objects. Multiple users can concurrently access an object, provided that the access is in a compatible mode. For example, multiple users can read the same row, but only one user is allowed to modify it at one time. So the blocking is normal and is expected in SQL Server. What we need to concern ourselves with is the excessive blocking that may impact the performance of the application. There can be multiple reasons for excessive blocking ranging from poorly configured hardware system for the workload, to the poor application design, and finally the design limitation in SQL

Server. Let us discuss these in some detail for a better understanding of actions to be taken to troubleshoot blocking issues.

The system configuration refers to the box (CPUs and memory), network, and I/O devices. Any bottleneck in these resources can impact the throughput of your system. When a user submits a request/batch, the SQL Server assigns a worker and schedules it on the CPU to execute it. If the number of incoming requests far exceeds the capacity of CPU(s) to process it, the end user may perceive that the request is blocked or running slow. Similarly, if SQL Server does not have enough memory, it can put pressure on the buffer pool to discard/flush pages that may be needed later or it can force compiled plans out of memory which in turn puts pressure on an I/O subsystem and on the CPU(s). So even if you have a perfectly designed application, the users may experience slowdown because of poorly configured hardware. However, as described earlier also, upgrading the hardware to meet the workload is useful, but it is not the panacea.

At a generic level, a database application allows multiple users to access and manipulate data in tables. Underneath, each user interaction is executed in the context of a transaction either started explicitly by the application or implicitly by the SQL Server. Transactions are considered to have four basic properties (Atomic, Consistent, Isolated, and Durable). Some of these properties (Atomic and Durable) are strictly enforced, however, there is some leeway in the other two depending on the isolation level of the transaction. For example, if a transaction is running under READ UNCOMMITTED isolation level, there is no guarantee that the data read is consistent or isolated from other concurrent transactions. As part of application design, you need to consider what transactional consistency can be tolerated by your application and then choose the transaction isolation level appropriately. For example, if as part of your application you are doing trend analysis using the sales data, it may be acceptable to run this transaction at a lower level of consistency. Similarly, you need to consider the database design and supporting indexes that best suit the demand of your application. It may sound like very straightforward and reasonable guidelines to design an application, but in reality, the applications are complex, and you may discover a lot more problems once the application has been deployed in production. The key is to understand what is causing the blocking in your environment, and then learn from them to improve the application in its next iteration. Increasingly, most customers buy prepackaged applications from Independent Software Vendors (ISVs), who may have developed the application initially for a different database platform and then ported to SQL Server. For example, if an application was developed for Oracle platform and it is now ported to SQL Server, it may encounter additional blocking because the default READ COMMITTED isolation level in SQL Server has been implemented using locks while the Oracle's implementation of read committed does not require locking. SQL Server 2005 addresses this issue by providing both a lock-based and row-version-based implementation of READ COMMITTED isolation level. Though you may not be able to change the application, you may be able to use information provided in this section for limited troubleshooting. For example, you may be able to add a missing index or prevent the lock escalation, and so on.

Detection of blocking

When thinking of blocking, most customers think of logical locks like share or exclusive locks on data rows, pages, and tables. Logical locks are the most visible and common blocking element in troubleshooting any blocking problems in database applications, but they are not all. As indicated earlier, the SQL Server employs a host of internal synchronization mechanisms to protect the integrity of internal structures. In this section, blocking will mean all kinds of blocking, including logical locks. At any given time in your workload, you need to see if you are getting significantly more blocking than expected (i.e., compared to your baseline). Starting with SQL Server 2005, the DMV sys.dm_os_wait_stats can be used see the cumulative waits that the threads or workers have encountered since the SQL Server was started but you can reset them using DBCC SQLPERF ([sys.dm_os_wait_stats], clear). This DMV tracks over 200 different types of waits and by looking at these wait statistics, you can identify the wait types that are causing more blocking in your application. These wait types include waits for logical locks, waits for I/O, waits for memory grant, and more. Since these wait statistics are cumulative, you can poll this DMV at regular intervals and identify the total waits, in milliseconds, for each wait type in the polling interval. Using this information, you can then probe into the cause of the waits or blocking.

The following DMV query shows the top-10 waits encountered in your application.

```
SELECT TOP 10
    wait_type,
    waiting_tasks_count AS tasks,
    wait_time_ms,
    max_wait_time_ms AS max_wait,
    signal_wait_time_ms AS signal
FROM sys.dm_os_wait_stats
ORDER BY wait_time_ms DESC
```

Here is the output of this DMV:

wait_type	tasks	wait_time_ms	max_wait	signal
LAZYWRITER_SLEEP	166202	344796328	55732593	21578
SQLTRACE_BUFFER_FLUSH	41562	344794437	55733359	4546
LCK_M_S	587	492205	149031	93
IO_COMPLETION	14462	68703	609	265
PAGEIOLATCH_SH	3150	51281	1328	359
ASYNC_NETWORK_IO	2525	34187	2015	1500
SOS_SCHEDULER_YIELD	87964	32687	3734	32687
BROKER_TASK_STOP	9	20359	10000	0
SLEEP_TASK	295304	18937	1015	10812
PAGEIOLATCH_EX	1560	16171	515	46

Here is a brief description of some of the waits in the output above:

- **LCK_M_S:** This wait occurs when a task is waiting to acquire a shared lock. In this case, you see the maximum a task waited to get the shared lock is 149,031 milliseconds. Another useful information is that a total of 587 tasks waited to get the SH lock.
- **LAZYWRITER_SLEEP:** This represents the wait by lazywriter thread. You may recall that lazywriter thread wakes up periodically and writes dirty pages to the disk. So the waits encountered by lazywriter thread are normal.
- **PAGEIOLATCH_SH:** This wait occurs when a task is waiting on a latch for a buffer that is in an I/O request. The latch request is in shared mode. Long waits of this type can indicate a problem with the disk subsystem.
- **PAGEIOLATCH_EX:** This wait occurs when a task is waiting on a latch for a buffer that is not in an I/O request. The latch request is in exclusive mode.
- The signal wait is the time between when a worker has been granted access to the resource and the time it gets scheduled on the CPU. A long signal wait may imply high CPU contention.

Please refer to BOL for the description of other wait types. Once you have identified that your application is encountering more blocking or waits compared to the baseline, the next step is to identify what is causing it.

System Monitor counters also provides a rich set of counters to detect blocking. Some of the key counters are:

- **SQLServer:Locks: Average Wait Time (ms)**: represents the average wait time (milliseconds) for each lock request that resulted in a wait.
- **SQLServer:Locks: Lock Requests/Sec**: represents the number of new locks and lock conversions requested from the lock manager.
- **SQLServer:Locks: Lock Wait Time (ms)**: represents total wait time (milliseconds) for locks in the last second.

- SQLServer:Locks: Lock Waits/Sec represents number of lock requests that could not be satisfied immediately and required the caller to wait before being granted the lock.
- SQLServer:Locks: Number of Deadlocks/Sec represents the number of lock requests that resulted in a deadlock.
- SQLServer:General Statistics: Processes Blocked represents the number of currently blocked processes.
- SQLServer:Access Methods: Table Lock Escalations/Sec represents the number of times locks were escalated to table-level granularity.

In earlier sections, we have already discussed how to isolate and troubleshoot waits related to CPU, I/O, and memory and waits incurred because of contention in tempdb. The following section focuses on the waits caused by logical locks in the application.

Isolating and troubleshooting blocking problems

As we outlined earlier, blocking is normal and expected in any concurrent applications. Significant blocking is often the result of long-running transactions and the isolation levels they are running at. When a transaction modifies a row, it holds an X lock on the row for the duration of the transaction. Similarly, if the transaction is running in higher isolations levels (that is, repeatable read and serializable), even the shared locks are held for the duration of the transactions. Some generic guidelines to troubleshoot blocking problems are:

- Shorten the duration of the transaction and run it at a lower isolation level. Avoid user interactions within a transaction. User interactions can potentially take a long unbounded amount of time. This can hold up the crucial resources in your application and deny other transactions from proceeding.
- Minimize the data that needs to be accessed by the transaction. Sometimes, if a useful index is missing, it may force transaction to read data that is really not needed.
- When doing DML operations on objects, try designing your application in such a way that you access objects in the same order. Imagine if you have two stored procedure such that one stored procedure updates table t1 and then table t2 within the same transaction, while the other stored procedure updates table t2 first and then table t1 in another transaction. This can potentially lead to a deadlock.
- If you are doing DML operations on a large number of rows, break it into smaller transactions to prevent lock escalation.

To troubleshoot blocking problems, you will need to know which transactions are holding locks and which transactions are blocked. You can find out at any given times all the locks that have been granted or waited upon by currently executing transactions using the following DMV query. This query provides an output similarly to the stored procedure sp_lock.

```

SELECT
    request_session_id AS spid,
    resource_type AS rt,
    resource_database_id AS rdb,
    (CASE resource_type
        WHEN 'OBJECT' then object_name(resource_associated_entity_id)
        WHEN 'DATABASE' THEN ''
        ELSE (SELECT object_name(object_id)
              FROM sys.partitions
              WHERE hobt_id=resource_associated_entity_id)
    END) AS objname,
    resource_description AS rd,
    request_mode AS rm,
    request_status AS rs
FROM sys.dm_tran_locks

```

Here is the sample output of this query that shows session 56 is blocked by session 53, which holds an X lock on RID 1:143:3.

Code View:

spid	rt	rdb	objname	rd	rm	rs
56	DATABASE	9			S	GRANT
53	DATABASE	9			S	GRANT
56	PAGE	9	t_lock	1:143	IS	GRANT
53	PAGE	9	t_lock	1:143	IX	GRANT
53	PAGE	9	t_lock	1:153	IX	GRANT
56	OBJECT	9	t_lock		IS	GRANT
53	OBJEC	9	t_lock		IX	GRANT
53	KEY	9	t_lock	(a400c34cb	X	GRANT
53	RID	9	t_lock	1:143:3	X	GRANT
56	RI	9	t_lock	1:143:3	S	WAIT

If you want more details, you can combine the sys.dm_tran_locks DMV with other DMVs to get more detailed information, such as duration of the block and the Transact-SQL statement being executed by the blocked transaction, as follows:

Code View:

```

SELECT
    t1.resource_type,
    'database' = DB_NAME(resource_database_id),
    'blk object' = t1.resource_associated_entity_id,
    t1.request_mode,
    t1.request_session_id,
    t2.blocking_session_id,
    t2.wait_duration_ms,
    (SELECT SUBSTRING(text, t3.statement_start_offset/2 + 1,
    (CASE WHEN t3.statement_end_offset = -1
        THEN LEN(CONVERT(nvarchar(max),text)) * 2
        ELSE t3.statement_end_offset
    END - t3.statement_start_offset)/2)
    FROM sys.dm_exec_sql_text(sql_handle)) AS query_text,
    t2.resource_description
FROM
    sys.dm_tran_locks AS t1,
    sys.dm_os_waiting_tasks AS t2,
    sys.dm_exec_requests AS t3
WHERE
    t1.lock_owner_address = t2.resource_address AND
    t1.request_request_id = t3.request_id AND
    t2.session_id = t3.session_id
  
```

Here is a sample output of this query where one transaction holds an X lock on the object while another transaction is waiting to read it. The output is only one row, but we have wrapped it to keep the width manageable.

Code View:

resource_type	database	blk object	request_mode

RID	foo	72057594039631872	S
request_session_id	blocking_session_id		
-----	-----	-----	-----
54	53	7585953	
query_text	resource_description		
-----	-----	-----	-----
SELECT * FROM [t1]	ridlock fileid=1 pageid=167 dbid=7		
WHERE [c1]=@1	id=lock36673c0 mode=X		
	associatedObjectId=72057594039631872		

As you can see here, the session requesting the lock was executing a SELECT on the table t1 and has requested a shared lock (S) on a row. Since another transaction holds an exclusive (X) lock on the row, the S lock cannot be granted and the requestor must wait. This is a classic writer-reader blocking scenario. In this example output, you can notice that the transaction has been waiting rather a long time, for a total of 7,585,953 milliseconds. This is also an indication that the blocking transaction is running for a long time, and you need to examine why it is so. Other ways to troubleshoot this problem could be to execute the SELECT under READ UNCOMMITTED isolation level, or you can troubleshoot readers/writers blocking by either using the new SNAPSHOT ISOLATION (SI) level or by enabling the READ_COMMITTED_SNAPSHOT (RCSI) at the database level. It provides a row-versioning-based nonblocking implementation of read-committed isolation level. Note, that using SI will require an application change, while RCSI will only be useful if the reader was running under READ COMMITTED isolation level. Note that these isolation levels do help, but there may be other reasons like missing an index, which may cause the writer or reader transaction to acquire a coarser granularity lock.

More commonly, blocking is often related to a few objects. The sys.dm_db_index_operational_stats DMV function provides comprehensive index usage statistics including blocking experienced while accessing that index. You can invoke this function at database level, individual table or index, or even partition level. In terms of blocking, it provides a detailed accounting of locking statistics per table, index, and partition as described below. Some of the information provided by this DMV is as follows:

- Row and Page lock counts incurred while accessing this index. A high number here indicates that this index is heavily used and can possibly have locking contention.
- Number of times a request had to wait to acquire a row or a page lock. Again, a large number here implies that there is, in fact, a contention in accessing this index.
- Number of times the SQL Server escalated the lock on this index to the Table level.
- Number of inserts, deletes, and updates on the leaf nodes of this index.

This information is cumulative from instance start-up, the information is not retained across instance restarts, and there is no way to explicitly reset it. The data returned by this DMV exists only as long as the metadata cache object representing the heap or index is available. The values for each column are set to zero whenever the metadata for the heap or index is brought into the metadata cache and statistics are accumulated until the cache object is removed from the metadata cache. However, you can periodically poll this table and collect this information in a table that can be queried further. One of the key things to watch out for here is what kinds of operations are being done on the index. If the index is rarely used for singleton or range scans, then this index is not very useful and should probably be removed. Similarly, if there are one or missing indexes, it may overload some other scan path (i.e. index). The following DMV query shows the operational statistics on all the indexes on a table called employee:

```
SELECT index_id,
       range_scan_count,
       row_lock_count,
       page_lock_count
```

```
FROM sys.dm_db_index_operational_stats(DB_ID ('<db-name>'),
OBJECT_ID('employee'), NULL, NULL)
```

Here is the output from this query, which shows that the nonclustered index (index_id = 2) has only been used only once so far. This is not an issue if this table is mostly read-only but if you perform a significant number of INSERT, DELETE, or UPDATE operations on the table then this index should probably be removed. Note, anytime you do any INSERT and DELETE operations on a table, you will need to insert or remove rows from all indexes defined on the table. For UPDATE operations, you will only need to modify an index if one or more of its key or included columns are being updated.

Code View:

index_id	range_scan_count	row_lock_count	page_lock_count
1	1500	1025500	92780
2	1	100	20

To detect missing indexes, please refer to the I/O bottleneck section.

Two other common blocking issues that need special attention are deadlocks and lock escalations.

Deadlocks

Deadlocks occur when two or more transactions or tasks are waiting on each other to acquire the resources needed to complete. SQL Server can detect deadlock on many different resource types. For example, a resource can be a logical lock, a memory grant, or a worker thread. However, normally when a transaction T1 requests a lock to access a resource, it may get blocked because some other transaction, T2, may have already locked the resource in a conflicting mode. In that case, T1 waits for the lock to be released before proceeding. Under normal operations, once T2 releases the lock, it is then acquired by T1. Two blocking scenarios are of interest here. First, what if T2 never releases the lock? In this case, unless a lock time-out is specified, T1 will wait indefinitely. An external intervention is required to break it, typically by killing the first transaction.

Second, what if T2 requests a lock on another resource that is already locked by T1 in conflicting mode? Since T1 is waiting for T2, and now T2 is waiting for T1, it results in a deadlock, and in this case, no amount of waiting will help. SQL Server includes logic to detect such cycles in resource through a background task, called the lock monitor, which checks for cycles periodically. The frequency is usually every 5 seconds but it can vary depending how often deadlocks are detected. When a deadlock is discovered, SQL Server breaks the deadlock by selecting one of the processes as the victim. The victim is typically the process that has consumed to the lowest resources so far. The victim's session will receive a 1205 error and the transaction will be aborted. SQL Server provides you some control over which task will be chosen as deadlock victim, by means of deadlock priority session setting. When a session gets a 1205 error, it does not provide any information on the cause of why the deadlock happened or what sessions and resources were involved in the deadlock cycle. Without this information, it is difficult to understand the causes of deadlock and to prevent or minimize it in the future. To get the detailed diagnostic information on the deadlock, it is recommended that you run SQL Server with traceflag 1222 (new in SQL2005); this provides you more detailed information than the related traceflag 1204, which also can be used for a similar purpose.

More details about preventing, minimizing, and troubleshooting deadlocks can be found in [Chapter 6](#).

Lock escalation

When an application acquires many locks at ROW or PAGE granularity, SQL Server may decide to escalate these locks to Table level to reduce the memory needed for the locks and to minimize the overhead of acquiring and releasing locks. There are primarily two conditions for lock escalation. First, SQL Server triggers a lock escalation when the number of locks held (note that it is different from acquired) by a statement on an index or a heap within a statement exceeds a threshold (currently set to approximately 5,000). These locks include the intent locks as well. Second, SQL Server triggers lock escalation when the memory taken by lock resources greater than 40 percent of the non-AWE (32 bit) or regular (64 bit) enabled memory when the locks configuration option is set to 0, the default value. In this case, the lock memory is allocated dynamically as needed. If the lock has been configured, then lock escalation is triggered once the memory taken by lock resources is greater than 40 percent of the configured memory of locks (i.e., when a nonzero value for the locks configuration option). When the locks configuration option is used, the locks memory is statically allocated when SQL Server starts.

When lock escalation is triggered, SQL Server attempts to escalate the locks to Table level, but the attempt may fail if there are conflicting locks. For example, if S locks need to be escalated to Table level and there are concurrent X locks on one or more rows/pages of the target table, the lock escalation attempt will fail. However, SQL Server periodically, for every 1,250 new locks acquired by the lock owner (that is, transaction), attempts to escalate the lock. If the lock escalation succeeds, the SQL Server releases the lower granularity locks, and the associated lock memory, on the index or the heap. A successful lock escalation can potentially lead to blocking (because at the time of lock escalation, there cannot be any conflicting access) of future concurrent access to the index or the heap by transactions in conflicting lock mode. So the lock escalation, though it reduces the number of locks takes, is not always a good idea for all applications.

You can disable lock escalation by using two trace flags as follows:

- Traceflag 1211 It disables lock escalation at the current threshold (5,000) on a per index/ heap, per statement basis. When this traceflag is in effect, the locks are never escalated. It also instructs SQL Server to ignore the memory acquired by the lock manager up to a maximum statically allocated lock memory or 60 percent of non-AWE(32 bit)/regular(64 bit) of the dynamically allocated memory. At this time, out-of-lock memory error is generated. This can potentially be damaging as a misbehaving application can exhaust SQL Server memory by acquiring a large number of locks. This, in the worst case, can stall the SQL Server or degrade its performance to an unacceptable level. For these reasons, a caution must be exercised when using this traceflag.
- Traceflag 1224 This traceflag is similar to traceflag 1211, with one key difference. It enables lock escalation when lock manager acquires 40 percent of the statically allocated memory or (40 percent) non-AWE (32 bit)/regular (64 bit) dynamically allocated memory. Additionally, if this memory cannot be allocated due to other components taking up more memory, the lock escalation can be triggered earlier. SQL Server will generate an out-of-memory error when memory allocated to lock manager exceeds the statically allocated memory or 60 percent of non-AWE (32 bit)/regular memory for dynamic allocation.

More details about preventing, minimizing and troubleshooting lock escalation can be found in [Chapter 6](#).

Summary

In this chapter, we discussed a generic strategy on how to diagnose and troubleshoot common performance problems in your database application running on SQL Server. Given the large number of possible applications with all their varied workloads and hardware configurations that can be deployed on SQL Server, it is nearly impossible to foresee all performance problems and provide an absolutely comprehensive set of troubleshooting guidelines. Our hope is that what we have described so far can give you a good idea of some key System Monitor counters and DMVs to get you started. As your experience with the application grows,

you will develop your own set of data that needs to be monitored. But whatever data that you collect, it is important to know what the baseline for your workload is. Without a representative baseline of your workload, you are almost shooting in the dark when troubleshooting performance problems.

When using DMV queries, you will need to get better understanding of SQL Server architecture to understand DMVs better. Armed with that knowledge, you will be able to create your own DMV queries that provide you the most useful information.

You may also want to refer to a tool that Microsoft will be releasing after this book goes to print, but will be available on the companion Web site that is based on DMVs and can be used to troubleshoot some common performance problems. The basic idea of this tool is to poll key DMVs at regular intervals and load that data into a database, which can be used as a performance warehouse. Now you can use the power of SQL Server to analyze this data. This tool is built using reports starting with a top-level view of your SQL Server and then provides you the capability to drill down into specific problems right down to the query level. The power of this tool is that it hides the complexity of data monitoring and analyzing from users and lets them focus on the performance troubleshooting issues.

Chapter 2. Tracing and Profiling

by Adam Machanic

In this chapter:	
SQL Trace Architecture and Terminology	57
Security and Permissions	59
Getting Started: Profiler	61
Server-Side Tracing and Collection	69
Troubleshooting and Analysis with Traces	79
Tracing Considerations and Design	96
Auditing: SQL Server's Built-in Traces	100
Summary	102

Query tuning, optimization, and general troubleshooting are all made possible through visibility into what's going on within SQL Server; it would be impossible to fix problems without being able to identify what caused them. SQL Trace and SQL Server Profiler, some of the most powerful tools provided by SQL Server, can give you a real-time or near real-time peek into exactly what the database engine is up to, at a very granular level.

Included in the tracing toolset are over 170 events that you can monitor and filter to get a look at anything from a broad overview of user logins down to such fine-grained information as the lock activity done by a specific server process id (spid). This data is all made available via a specialized user interface tool, SQL Server Profiler, in addition to a series of server-side stored procedures and .NET classes, giving you the flexibility to roll a custom solution when a problem calls for one.

This chapter outlines the overall architecture used by the system under the covers, then goes on to show you how to actually use SQL Trace to solve some common problems. SQL Trace is fairly easy to use once you get familiar with it, but the combination of possible options is extensive, so we'll focus on those that you'll use

most often in the real world. We'll also spend some time discussing how best to design your traces so as to keep things as efficient as possible. Tracing server activity bears a slight performance penalty, but by paying attention to best practices you can help minimize any impact.

SQL Trace Architecture and Terminology

SQL Trace is a SQL Server database engine technology, and it is important to understand that the client-side Profiler tool is really nothing more than a wrapper over the server-side functionality. When tracing, we monitor for specific events, which are generated when various actions occur in the database engine. For example, a user login or the execution of a query are each actions that cause events to fire. Each event has an associated collection of columns, which are attributes that contain data collected when the event fires. For instance, in the case of a query we can collect data about when the query started, how long it took, and how much CPU time it used. Finally, each trace can specify filters, which limit the results returned based on a set of criteria. One could, for example, specify that only events that took longer than 50 milliseconds should be returned.

With over 170 events and 65 columns to choose from, the number of data points that can be collected is quite large. Not every column can be used with every event, but the complete set of allowed combinations is almost 4,000. Thinking about memory utilization to hold all of this data and the processor time needed to create it, you might be interested in how SQL Server manages to keep itself running efficiently while generating so much information. The answer is that SQL Server doesn't actually collect any data at all until someone asks for it—the model instead is to selectively enable collection only as necessary.

Internal Trace Components

The central component of the SQL Trace architecture is the trace controller, which is a shared resource that manages all traces created by any consumer. Throughout the database engine are various event producers; for example, they are found in the query processor, lock manager, and cache manager. Each of these producers is responsible for generating events that pertain to certain categories of server activity, but each of the producers are disabled by default, and therefore generate no data. When a user requests that a trace be started for a certain event, a global bitmap in the trace controller is updated, letting the event producer know that at least one trace is listening, and causing the event to begin firing. Managed along with this bitmap is a secondary list of which traces are monitoring which events.

Once an event fires, its data is routed into a global event sink, which queues the event data for distribution to each trace that is actively listening. The trace controller routes the data to each listening trace based on its internal list of traces and watched events. In addition to the trace controller's own lists, each individual trace also keeps track of which events it is monitoring, along with which columns are actually being used as well as what filters are in place. The event data returned by the trace controller to each trace is filtered, and the data columns are trimmed down as necessary, before the data is routed to an I/O provider.

Trace I/O Providers

The trace I/O providers are what actually send the data along to its final destination. The available output formats for trace data are either a file on the database server (or a network share) or a rowset to a client. Both providers use internal buffers to ensure that if the data is not consumed quickly enough (that is, written to disk or read from the rowset) that it will be queued. However, there is a big difference in how the providers handle a situation in which the queue grows beyond a manageable size.

The file provider is designed with a guarantee that no event data will be lost. To make this work even if an I/O slowdown or stall occurs, the internal buffers begin to fill if disk writes are not occurring quickly enough.

Once the buffers fill up, threads sending event data to the trace begin waiting for buffer space to free up. In order to avoid threads waiting on trace buffers, it is imperative to ensure that tracing is done to a fast enough disk system. To monitor for these waits, watch the SQLTRACE_LOCK and IO_COMPLETION wait types.

The rowset provider, on the other hand, is not designed to make any data loss guarantees. If data is not being consumed quickly enough and its internal buffers fill, it waits up to 20 seconds before it begins jettisoning events in order to free buffers to get things moving. The SQL Server Profiler client tool will send a special error message if events are getting dropped, but you can also find out if you're headed in that direction by monitoring SQL Server's TRACEWRITE wait type, which is incremented as threads are waiting for buffers to free up.

A background trace management thread is also started whenever at least one trace is active on the server. This background thread is responsible for flushing file provider buffers (done every 4 seconds), in addition to closing rowset-based traces that are considered to be expired (this occurs if a trace has been dropping events for more than 10 minutes). By flushing the file provider buffers only occasionally, rather than writing the data to disk every time an event is collected, SQL Server can take advantage of large block writes, dramatically reducing the overhead of tracing, especially on extremely active servers.

A common question asked by DBAs new to SQL Server is why no provider exists that can write trace data directly to a table. The reason for this limitation comes down to the amount of overhead that would be required. Because a table does not support large block writes, SQL Server would have to write the event data row by row. The performance degradation caused by event consumption would require either dropping a lot of events or, if a lossless guarantee were enforced, causing a lot of blocking to occur. Neither scenario is especially palatable, so SQL Server simply does not provide this ability. However, as we will see later in the chapter, it is easy enough to load the data into a table either during or after tracing, so this is not much of a limitation.

Chapter 2. Tracing and Profiling

â Adam Machanic

In this chapter:	
SQL Trace Architecture and Terminology	57
Security and Permissions	59
Getting Started: Profiler	61
Server-Side Tracing and Collection	69
Troubleshooting and Analysis with Traces	79
Tracing Considerations and Design	96
Auditing: SQL Server's Built-in Traces	100
Summary	102

Query tuning, optimization, and general troubleshooting are all made possible through visibility into what's going on within SQL Server; it would be impossible to fix problems without being able to identify what caused them. SQL Trace and SQL Server Profiler, some of the most powerful tools provided by SQL Server, can give you a real-time or near real-time peek into exactly what the database engine is up to, at a very granular level.

Included in the tracing toolset are over 170 events that you can monitor and filter to get a look at anything from a broad overview of user logins down to such fine-grained information as the lock activity done by a specific server process id (spid). This data is all made available via a specialized user interface tool, SQL Server Profiler, in addition to a series of server-side stored procedures and .NET classes, giving you the flexibility to roll a custom solution when a problem calls for one.

This chapter outlines the overall architecture used by the system under the covers, then goes on to show you how to actually use SQL Trace to solve some common problems. SQL Trace is fairly easy to use once you get familiar with it, but the combination of possible options is extensive, so we'll focus on those that you'll use most often in the real world. We'll also spend some time discussing how best to design your traces so as to keep things as efficient as possible. Tracing server activity bears a slight performance penalty, but by paying attention to best practices you can help minimize any impact.

SQL Trace Architecture and Terminology

SQL Trace is a SQL Server database engine technology, and it is important to understand that the client-side Profiler tool is really nothing more than a wrapper over the server-side functionality. When tracing, we monitor for specific events, which are generated when various actions occur in the database engine. For example, a user login or the execution of a query are each actions that cause events to fire. Each event has an associated collection of columns, which are attributes that contain data collected when the event fires. For instance, in the case of a query we can collect data about when the query started, how long it took, and how much CPU time it used. Finally, each trace can specify filters, which limit the results returned based on a set of criteria. One could, for example, specify that only events that took longer than 50 milliseconds should be returned.

With over 170 events and 65 columns to choose from, the number of data points that can be collected is quite large. Not every column can be used with every event, but the complete set of allowed combinations is almost 4,000. Thinking about memory utilization to hold all of this data and the processor time needed to create it, you might be interested in how SQL Server manages to keep itself running efficiently while generating so much information. The answer is that SQL Server doesn't actually collect any data at all until someone asks for it—the model instead is to selectively enable collection only as necessary.

Internal Trace Components

The central component of the SQL Trace architecture is the trace controller, which is a shared resource that manages all traces created by any consumer. Throughout the database engine are various event producers; for example, they are found in the query processor, lock manager, and cache manager. Each of these producers is responsible for generating events that pertain to certain categories of server activity, but each of the producers are disabled by default, and therefore generate no data. When a user requests that a trace be started for a certain event, a global bitmap in the trace controller is updated, letting the event producer know that at least one trace is listening, and causing the event to begin firing. Managed along with this bitmap is a secondary list of which traces are monitoring which events.

Once an event fires, its data is routed into a global event sink, which queues the event data for distribution to each trace that is actively listening. The trace controller routes the data to each listening trace based on its internal list of traces and watched events. In addition to the trace controller's own lists, each individual trace also keeps track of which events it is monitoring, along with which columns are actually being used as well as what filters are in place. The event data returned by the trace controller to each trace is filtered, and the data columns are trimmed down as necessary, before the data is routed to an I/O provider.

Trace I/O Providers

The trace I/O providers are what actually send the data along to its final destination. The available output formats for trace data are either a file on the database server (or a network share) or a rowset to a client. Both providers use internal buffers to ensure that if the data is not consumed quickly enough (that is, written to disk or read from the rowset) that it will be queued. However, there is a big difference in how the providers handle a situation in which the queue grows beyond a manageable size.

The file provider is designed with a guarantee that no event data will be lost. To make this work even if an I/O slowdown or stall occurs, the internal buffers begin to fill if disk writes are not occurring quickly enough. Once the buffers fill up, threads sending event data to the trace begin waiting for buffer space to free up. In order to avoid threads waiting on trace buffers, it is imperative to ensure that tracing is done to a fast enough disk system. To monitor for these waits, watch the SQLTRACE_LOCK and IO_COMPLETION wait types.

The rowset provider, on the other hand, is not designed to make any data loss guarantees. If data is not being consumed quickly enough and its internal buffers fill, it waits up to 20 seconds before it begins jettisoning events in order to free buffers to get things moving. The SQL Server Profiler client tool will send a special error message if events are getting dropped, but you can also find out if you're headed in that direction by monitoring SQL Server's TRACEWRITE wait type, which is incremented as threads are waiting for buffers to free up.

A background trace management thread is also started whenever at least one trace is active on the server. This background thread is responsible for flushing file provider buffers (done every 4 seconds), in addition to closing rowset-based traces that are considered to be expired (this occurs if a trace has been dropping events for more than 10 minutes). By flushing the file provider buffers only occasionally, rather than writing the data to disk every time an event is collected, SQL Server can take advantage of large block writes, dramatically reducing the overhead of tracing, especially on extremely active servers.

A common question asked by DBAs new to SQL Server is why no provider exists that can write trace data directly to a table. The reason for this limitation comes down to the amount of overhead that would be required. Because a table does not support large block writes, SQL Server would have to write the event data row by row. The performance degradation caused by event consumption would require either dropping a lot of events or, if a lossless guarantee were enforced, causing a lot of blocking to occur. Neither scenario is especially palatable, so SQL Server simply does not provide this ability. However, as we will see later in the chapter, it is easy enough to load the data into a table either during or after tracing, so this is not much of a limitation.

Security and Permissions

Tracing can expose a lot of information about not only the state of the server, but also the data sent to and returned from the database engine by users. The ability to monitor individual queries down to the batch or even query plan level is at once both powerful and worrisome; even exposure of stored procedure input arguments can give an attacker a lot of information about the data in your database.

In order to protect SQL Trace from users that should not be able to view the data it exposes, previous versions of SQL Server allowed only administrative users (members of the sysadmin fixed server role) access to start traces. That restriction proved a bit too inflexible for many development teams, and as a result it has been loosened.

ALTER TRACE Permission

In SQL Server 2005, a new permission exists, called ALTER TRACE. This is a server-level permission (granted to a login principal), and allows access to start, stop, or modify a trace, in addition to being able to generate user-defined events (more on this later in the chapter, in the "[Stored Procedure Debugging](#)" section).

Keep in mind that this permission is granted at the server level, and access is at the server level; if a user can start a trace, he or she can retrieve event data no matter what database the event was generated in. The inclusion of this permission in SQL Server is a great step in the right direction for situations in which developers might need to run traces on production systems in order to debug application issues, but it's important to not grant this permission too lightly. It's still a potential security threat, even if it's not nearly as severe as giving someone full sysadmin access.

To grant ALTER TRACE permission to a login, use the GRANT statement as follows (in this example, the permission is granted to a server principal called "Jane"):

```
GRANT ALTER TRACE TO Jane;
```

Protecting Sensitive Event Data

In addition to being locked down so that only certain users can use SQL Trace, the tracing engine itself has a couple of built-in security features to keep unwanted eyesâ off of private information. SQL Trace will automatically omit data if an event contains a call to a password-related stored procedure or statement. For example, a call to CREATE LOGIN including the WITH PASSWORD option will be blanked out by SQL Trace.

Note



In previous versions of SQL Server, SQL Trace automatically blanked out a query event if the string sp_password was found anywhere in the text of the query. This feature has been removed in SQL Server 2005, and you should not depend on it to protect your intellectual capital.

Another security feature of SQL Trace is knowledge of encrypted modules. SQL Trace will not return statement text or query plans generated within an encrypted stored procedure, user-defined function, or view. Again, this helps to safeguard especially sensitive data even from users who should have access to see traces.

Getting Started: Profiler

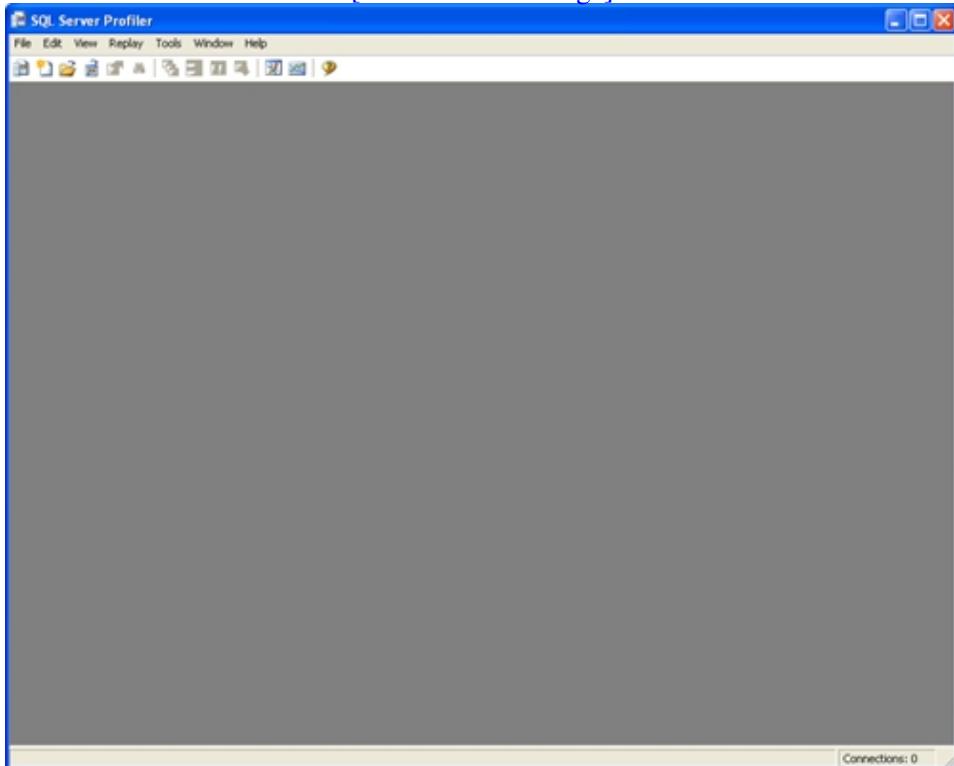
SQL Server 2005 ships with a powerful user interface tool that can be used to create, manipulate, and manage traces. SQL Server Profiler is the primary starting point for most tracing activity, and thanks to the ease with which it can help you get traces up and running, it is perhaps the most important SQL Server component available for quickly troubleshooting database issues. SQL Server Profiler also adds a few features to the toolset that are not made possible by SQL Trace itself. This section discusses those in addition to the base tracing capabilities.

Profiler Basics

The Profiler tool can be found in the Performance Tools subfolder of the SQL Server 2005 Start Menu folder. Once the tool is started, you will be greeted by a blank screen, as shown in [Figure 2-1](#). Click File, then New Trace . . . and connect to a SQL Server instance. You will be shown a Trace Properties dialog box with two tabs, General and Events Selection.

Figure 2-1. SQL Server Profiler main screen

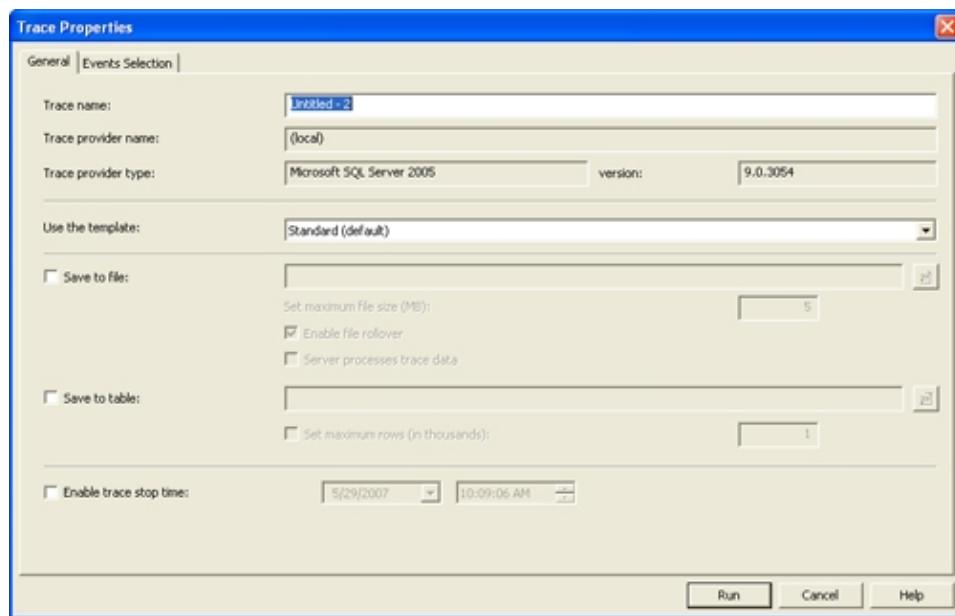
[[View full size image](#)]



The General tab, shown in [Figure 2-2](#), allows you to control how the trace will be processed by the consumer. The default setting is to use the rowset provider, displaying the events in real time in the SQL Server Profiler window. Also available are options to save the events to a file (on either the server or the client), or to a table. However, we generally recommend that you avoid these options on a busy server.

Figure 2-2. Choosing the I/O provider for the trace

[[View full size image](#)]



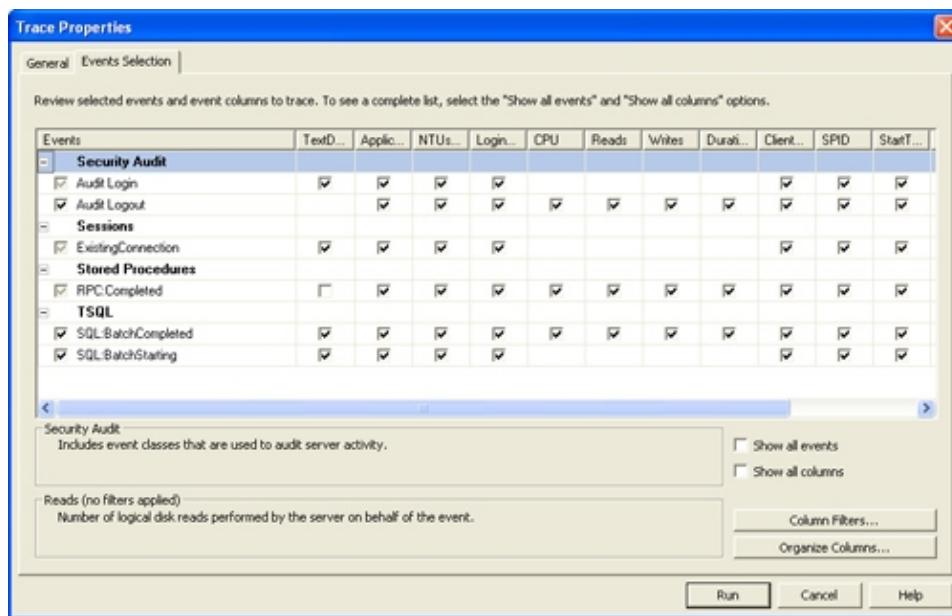
When you ask Profiler to save the events to a server-side file (done by selecting Server processes trace data), it actually starts two equivalent traces— one using the rowset provider and the other using the file provider. Two traces means twice as much overhead, and that is generally not a great idea. See the section "["Server-Side Tracing and Collection"](#)" later in this chapter for information on how to correctly set up a trace using the file provider in order to efficiently save to a server-side file. Saving to a client-side file does not use the file provider at all. Rather, the data is routed to the Profiler tool via the rowset provider, then saved from there to a file. This is more efficient than using Profiler to write to a server-side file, but you do incur network bandwidth because of using the rowset provider, and you also do not get the benefit of the lossless guarantee that the file provider offers.

Seeing the Save To Table option, you might wonder why our "["SQL Trace Architecture and Terminology"](#)" section stated that this is not an available feature of SQL Trace. The fact is, SQL Trace exposes no table output provider. Instead, when you use this option, the SQL Server Profiler tool uses the rowset provider and routes the data back into a table. If the table you save to is on the same server you're tracing, you can create quite a large amount of server overhead and bandwidth utilization, so if you must use this option we recommend saving the data to a table on a different server. SQL Server Profiler also provides an option to save the data to a table after you're done tracing, and this is a much more scalable choice in most scenarios.

The Events Selection tab, shown in [Figure 2-3](#), is where you'll spend most of your time configuring traces in SQL Server Profiler. This tab allows you to select events that you'd like to trace, along with associated data columns. The default options, shown in [Figure 2-3](#), collect data about any connections that exist when the trace starts (the ExistingConnection event) when a login or logout occurs (the Audit Login and Audit Logout events), when remote procedure calls complete (the RPC:Completed event), and when Transact-SQL batches start or complete (the SQL:BatchCompleted and SQL:BatchStarting events). By default, the complete list of both events and available data columns is hidden. Checking the Show All Events and Show All Columns check boxes brings the available selections into the UI.

Figure 2-3. Choosing event/column combinations for the trace

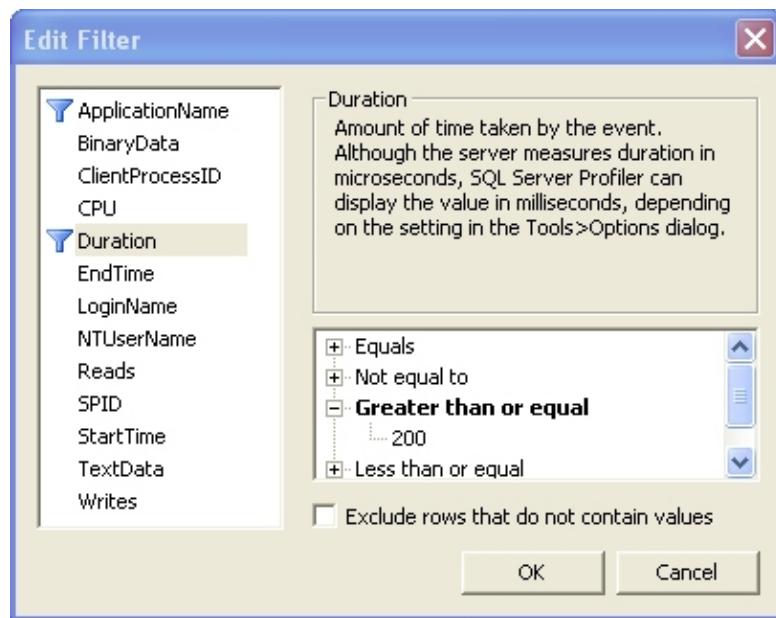
[\[View full size image\]](#)



These default selections are a great starting point and can be used as the basis for a lot of commonly required traces. The simplest questions that DBAs generally answer using SQL Trace are based around query cost and/or duration. What are the longest queries or the queries that are using the most resources? The default selections can help you answer those types of questions, but on an active server a huge amount of data would have to be collected. This would not only mean more work for you to be able to answer your question, but also more work for the server to collect and distribute that much data.

In order to narrow your scope and help ensure that tracing does not cause performance issues, SQL Trace offers the ability to filter the events based on various criteria. Filtration is exposed in SQL Server Profiler via the Column Filters... button in the Events Selection tab. Click this button to bring up an Edit Filter dialog box, similar to the one shown in [Figure 2-4](#). In this example, we only want to see events with a duration of greater than or equal to 200 ms. This is just an arbitrary number; an optimal choice should be discovered iteratively as you build up your knowledge of the tracing requirements for your particular application. Keep raising this number until you mostly receive only the interesting events (in this case, those with long durations) from your trace. By working this way, you can easily and quickly isolate the slowest queries in your system. See the "[Performance Tuning](#)" section later in this chapter for more information on this technique.

Figure 2-4. Defining a filter for events greater than 200 milliseconds



Tip



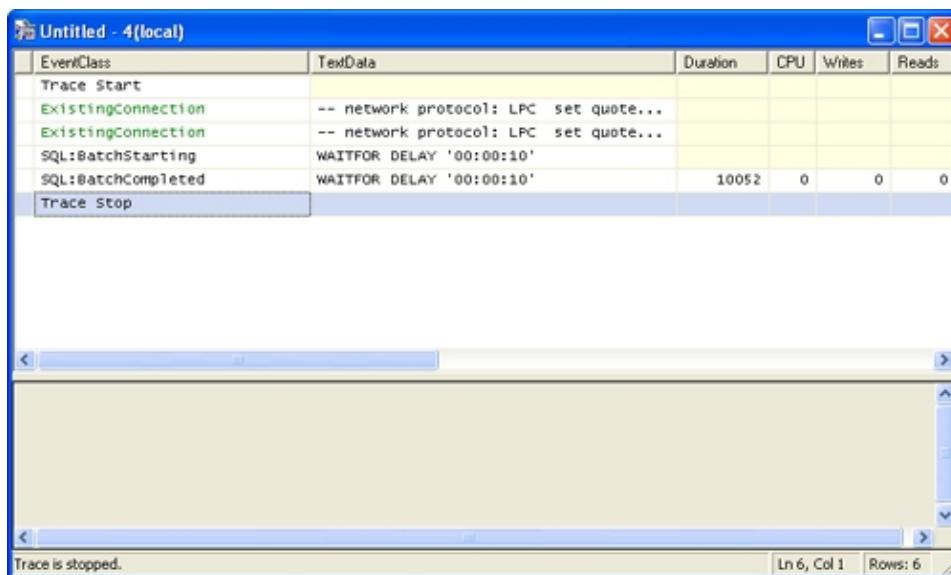
The list of data columns made available by SQL Server Profiler for you to use as a filter is the same list of columns available in the outer Events Selection UI. Make sure to check the Show All Columns checkbox in order to ensure that you see a complete list.

Once events are selected and filters are defined, the trace can be started. Click the Run button on the Trace Properties dialog box. Because Profiler uses the rowset provider, data will begin streaming back immediately. If you find that data is coming in too quickly for you to be able to read it, consider disabling auto scrolling using the Auto Scroll Window button on the SQL Server Profiler taskbar.

An important note on filters is that, by default, events that do not produce data for a specific column will not be filtered if a trace defines a filter for that column. For example, the SQL:BatchStarting event does not produce duration data—the batch is considered to start more or less instantly the moment it is submitted to the server. [Figure 2-5](#) shows a trace we ran with a filter the Duration column for values greater than 200. Notice that both the ExistingConnection and SQL:BatchStarting events are still returned even though they lack the Duration output column. To modify this behavior, check the Exclude Rows That Do Not Contain Values checkbox in the Edit Filter dialog, for the column you'd like to change the setting for.

Figure 2-5. By default, trace filters treat empty values as valid for the sake of the filter

[\[View full size image\]](#)



Saving and Replaying Traces

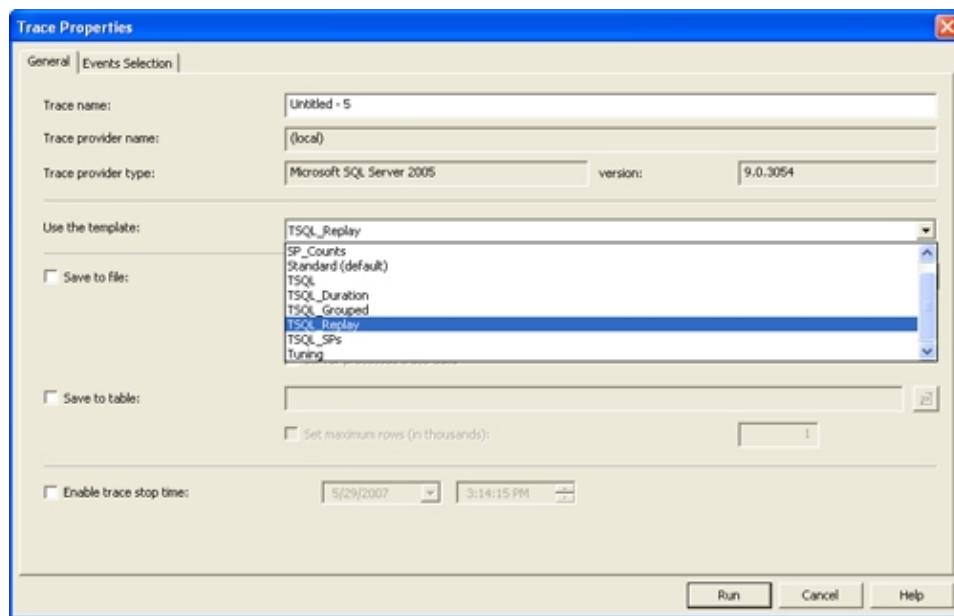
The functionality covered up through this point has all been made possible by SQL Server Profiler merely acting as a wrapper over what SQL Trace provides. In the "Server-Side Tracing and Collection" section later in this chapter, we will show you the mechanisms by which SQL Server Profiler does its work. But first we'll get into the features offered by SQL Server Profiler that make it more than a simple UI wrapper over the SQL Trace features.

When we discussed the General tab of the Trace Properties window earlier, we glossed over how the default events are actually set: They are included in the standard events template that ships with the product. A template is a collection of event and column selections, filters, and other settings that you can save to create reusable trace definitions. This can be an extremely useful feature if you do a lot of tracing; reconfiguring the options each time you need them is generally not a good use of your time.

In addition to the ability to save your own templates, Profiler ships with eight that have been predefined. Aside from the standard template that we already explored, one of the most important of these is the TSQL_Replay template, shown selected in [Figure 2-6](#). This template selects a variety of columns for 15 different events, each of which are required for Profiler to be able to Play Back (or Replay) a collected trace at a later time. By starting a trace using this template, then saving the trace data once collection is complete, you can do things such as use a trace as a test harness for reproducing a specific problem that might occur when certain stored procedures are called in the correct order.

Figure 2-6. Selecting the template

[\[View full size image\]](#)



To illustrate this functionality, we started a new trace using the TSQL_Replay template, and sent two batches from each of two connections, as shown in [Figure 2-7](#). The first spid (53, in the figure) selected 1, then the second spid (54) selected 2. Back to spid 53, which selected 3, and then finally back to spid 54, which selected 4. The most interesting thing to note in the figure is the second column, EventSequence. This column can be thought of almost like the IDENTITY property for a table. Its value is incremented globally as events are recorded by the trace controller, in order to create a single representation of the order in which events occurred in the server. This avoids problems that might occur when ordering by StartTime/EndTime (also in the trace, but not shown in the figure), as there will be no ties—the EventSequence will be unique per trace. The number is a 64-bit integer, and it is reset whenever the server is restarted, so it is unlikely that you will ever be able to trace enough to run it beyond its range.

Figure 2-7. Two spids sending interleaved batches

[\[View full size image\]](#)

EventClass	EventSequence	TextData	SPID	DatabaseName	ApplicationName
Trace Start					
ExistingConnection	2002	-- network protocol: LPC set quote...	52	master	Microsoft SQ...
ExistingConnection	2003	-- network protocol: LPC set quote...	53	master	Microsoft SQ...
ExistingConnection	2004	-- network protocol: LPC set quote...	54	master	Microsoft SQ...
SQL:BatchStarting	2007	SELECT 1	53	master	Microsoft SQ...
SQL:BatchCompleted	2008	SELECT 1	53	master	Microsoft SQ...
SQL:BatchStarting	2009	SELECT 2	54	master	Microsoft SQ...
SQL:BatchCompleted	2010	SELECT 2	54	master	Microsoft SQ...
SQL:BatchStarting	2011	SELECT 3	53	master	Microsoft SQ...
SQL:BatchCompleted	2012	SELECT 3	53	master	Microsoft SQ...
SQL:BatchStarting	2013	SELECT 4	54	master	Microsoft SQ...
SQL:BatchCompleted	2014	SELECT 4	54	master	Microsoft SQ...
Trace Stop					

Trace is stopped. [ln 13, Col 1] [Rows: 13]

Once the trace data has been collected, it must be saved and then reopened before a replay can begin. SQL Server Profiler offers the following options for saving trace data, which are available from the File menu:

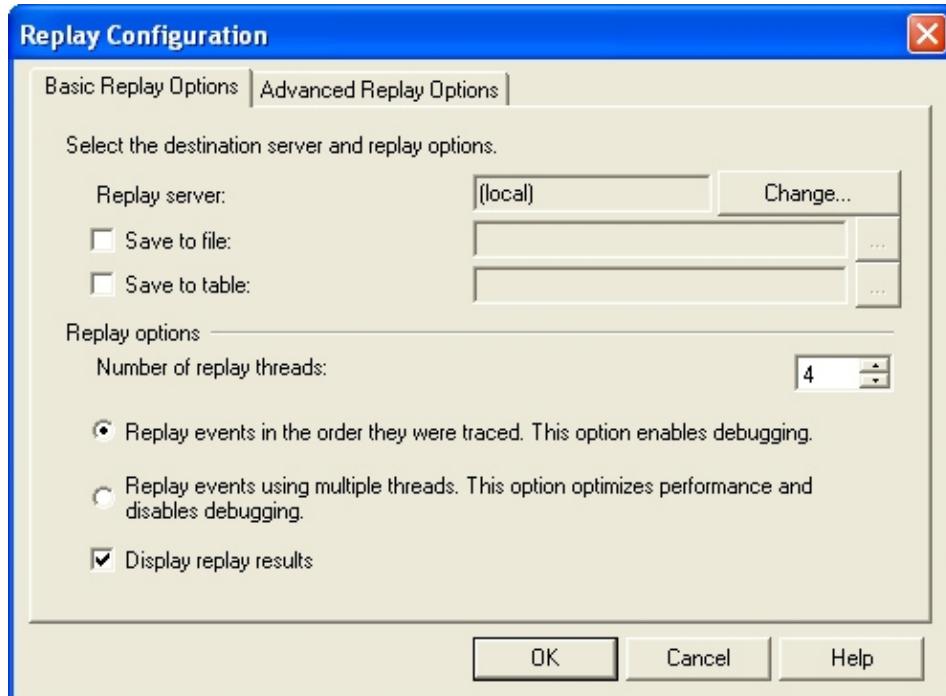
- The Trace File option is used to save the data to a file formatted using a proprietary binary format. This is generally the fastest way to save the data, and also the smallest in terms of bytes on disk.
- The Trace Table option is used to save the data to a new or previously created table in a database of your choosing. This option is useful if you need to manipulate or report on the data using T-SQL.
- The Trace XML File option saves the data to a text file formatted as XML.
- The Trace XML File For Replay option also saves the data to an XML text file, but only those events and columns needed for replay functionality are saved.

Any of these formats can be used as a basis from which to replay a trace, as long as you've collected all of the required events and columns needed to do a replay (guaranteed when you use the TSQL_Replay template). We generally recommend using the binary file format as a starting point, and saving to a table if manipulation using T-SQL is necessary. For instance, you might want to create a complex query that finds the top queries that use certain tables; something like that would be beyond the abilities of SQL Server Profiler. With regard to the XML file formats, so far, we have not found much use for them. But as more third-party tools hit the market that can use trace data, we may see more use cases.

Once the data has been saved to a file or table, the original trace window can be closed and the file or table reopened via SQL Server Profiler's File menu. Once a trace is reopened in this way, a Replay menu appears on the Profiler tool bar, allowing you to start replaying the trace, stop the replay, or set a breakpointâ useful when you want to test only a small portion of a larger trace.

After clicking Start, you will be asked to connect to a serverâ either the server you did the collection from, or another if you'd like to replay the same trace somewhere else. After connecting, the options dialog shown in [Figure 2-8](#) will be presented. The Basic Replay Options tab allows you to save results of the trace, in addition to modifying how the trace is played back.

Figure 2-8. Replay options dialog box



During the course of the replay, the same events used to produce the trace being replayed will be traced from the server on which you replay. The Save To File and Save To Table options are used for a client-side save. No server-side option exists for saving playback results.

The Replay options section is a bit confusing as worded. No matter which option you select, the trace will be replayed on multiple threads, corresponding to at most the Number Of Replay Threads specified. However, the Replay Events In The Order They Were Traced option ensures that all events will be played back in exactly the order in which they occurred, as based upon the EventSequence column. Multiple threads will still be used to simulate multiple spids. The Replay Events Using Multiple Threads option, on the other hand, allows SQL Server Profiler to reorder the order in which each spid starts to execute events, in order to enhance playback performance. Within a given spid, however, the order of events will remain consistent with the EventSequence.

To illustrate this difference, we replayed the trace shown in [Figure 2-7](#) twice, each using a different Replay option. [Figure 2-9](#) shows the result of the Replay In Order option, whereas [Figure 2-10](#) shows the result of the Multiple Threads option. In [Figure 2-9](#), the results show that the batches were started and completed in exactly the same order in which they were originally traced, whereas in [Figure 2-10](#) the two participating spids have each had all of their events grouped together rather than interleaved.

Figure 2-9. Replay using the In Order option

[[View full size image](#)]

(local).[master].[dbo].[inside_sql]

EventClass	EventSequence	TextData	ApplicationName	DatabaseName
SQL:BatchStarting	2007	SELECT 1	Microsoft SQ...	master
SQL:BatchCompleted	2008	SELECT 1	Microsoft SQ...	master
SQL:BatchStarting	2009	SELECT 2	Microsoft SQ...	master
SQL:BatchCompleted	2010	SELECT 2	Microsoft SQ...	master
SQL:BatchStarting	2011	SELECT 3	Microsoft SQ...	master
SQL:BatchCompleted	2012	SELECT 3	Microsoft SQ...	master

EventClass	TextData	SPID	IntegerData	DatabaseID	DatabaseName
ExistingConnection	-- network protocols: LPC set quote...	52	51	1	master
ExistingConnection	-- network protocols: LPC set quote...	53	55	1	master
ExistingConnection	-- network protocols: LPC set quote...	54	56	1	master
SQL:BatchStarting	SELECT 1	53	55	1	master
Replay Result Set Event		53	55	1	master
Replay Result Row Event	1	53	55	1	master
SQL:BatchStarting	SELECT 2	54	56	1	master
Replay Result Set Event		54	56	1	master
Replay Result Row Event	2	54	56	1	master
SQL:BatchStarting	SELECT 3	53	55	1	master
Replay Result Set Event		53	55	1	master

Figure 2-10. Replay using the Multiple Threads option

[[View full size image](#)]

(local).{master].[dbo].[inside_sq]					
EventClass	EventSequence	TextData	ApplicationName	DatabaseName	
SQL:BatchStarting	2007	SELECT 1	Microsoft SQ...	master	
SQL:BatchCompleted	2008	SELECT 1	Microsoft SQ...	master	
SQL:BatchStarting	2009	SELECT 2	Microsoft SQ...	master	
SQL:BatchCompleted	2010	SELECT 2	Microsoft SQ...	master	
SQL:BatchStarting	2011	SELECT 3	Microsoft SQ...	master	
SQL:BatchCompleted	2012	SELECT 3	Microsoft SQ...	master	
EventClass	TextData	SPID	IntegerData	DatabaseID	DatabaseName
ExistingConnection	-- network protocol: LPC set quote...	53	56	1	master
SQL:BatchStarting	SELECT 1	53	56	1	master
Replay Result Set Event		53	56	1	master
Replay Result Row Event	1	53	56	1	master
SQL:BatchStarting	SELECT 3	53	56	1	master
Replay Result Set Event		53	56	1	master
ExistingConnection	-- network protocol: LPC set quote...	52	51	1	master
ExistingConnection	-- network protocol: LPC set quote...	54	55	1	master
SQL:BatchStarting	SELECT 2	54	55	1	master
Replay Result Set Event		54	55	1	master
Replay Result Row Event	2	54	55	1	master

Ready. Rows: 1

The Multiple Threads option can be useful if you need to replay a lot of trace data where each spid has no dependency upon other spids. For example, this might be done in order to simulate, on a test server, a workload captured from a production system. The In Order option, on the other hand, is useful if you need to ensure that you can duplicate the specific conditions that occurred during the trace. For example, this might apply when debugging a deadlock or blocking condition that results from specific interactions of multiple threads accessing the same data.

SQL Server Profiler is a full-featured tool that provides extensive support for both tracing and doing simple work with trace data, but if you need to do advanced queries against your collected data or run traces against extremely active production systems, SQL Server Profiler falls short of the requirements. Again, SQL Server Profiler is essentially nothing more than a wrapper over functionality provided within the database engine, and instead of using it for all stages of the trace lifestyle, we can directly exploit the server-side tool to increase flexibility in some cases. In the following section, we'll show you how SQL Server Profiler works with the database engine to start, stop, and manage traces, and how you can harness the same tools for your needs.

Server-Side Tracing and Collection

Behind its nice user interface, SQL Server Profiler is nothing more than a fairly lightweight wrapper over a handful of system stored procedures which expose the true functionality of SQL Trace. In this section, we will explore which stored procedures are used, and how to harness SQL Server Profiler as a scripting tool rather than a tracing interface.

The following system stored procedures are used to define and manage traces:

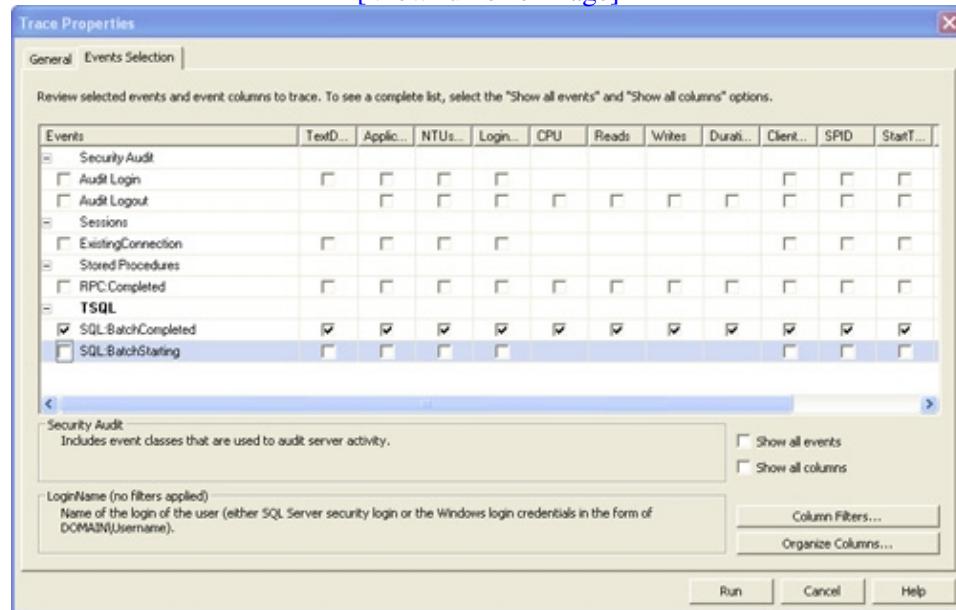
- `sp_trace_create` is used to define a trace and specify an output file location as well as other options that I'll cover in the coming pages. This stored procedure returns a handle to the created trace, in the form of an integer trace ID.
- `sp_trace_setevent` is used to add event/column combinations to traces based on the trace ID, as well as to remove them, if necessary, from traces in which they have already been defined.
- `sp_trace_setfilter` is used to define event filters based on trace columns.
- `sp_trace_setstatus` is called to turn on a trace, to stop a trace, and to delete a trace definition once you're done with it. Traces can be started and stopped multiple times over their lifespan.

Scripting Server-Side Traces

Rather than delve directly into the syntax specifications for each of the stored procedures— all of which are documented in detail in SQL Server Books Online—it is a bit more interesting to observe them in action. To begin, open up SQL Server Profiler, start a new trace with the default template, and unselect all events except for SQL:BatchCompleted, as shown in [Figure 2-11](#).

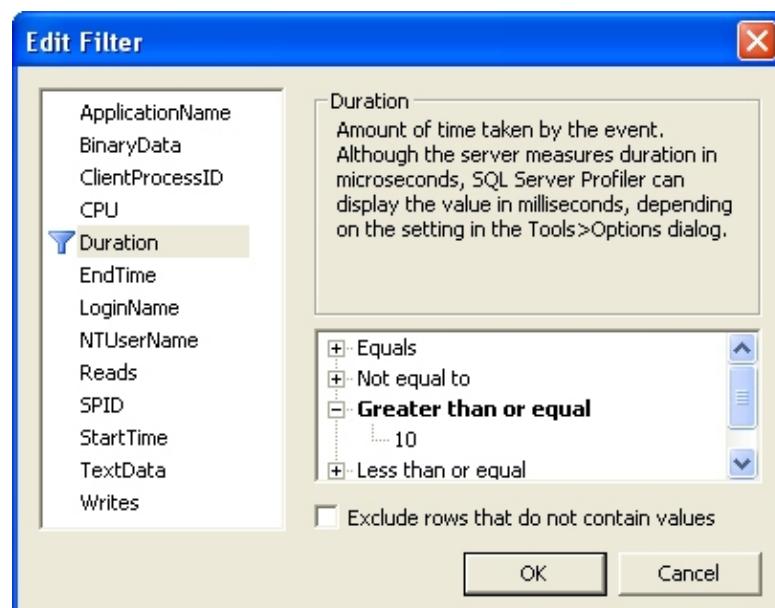
[Figure 2-11. Trace events with only selected](#)

[View full size image]



Next, remove the default filter on the ApplicationName column (set to not pick up SQL Server Profiler events), and add a filter on Duration, for greater than or equal to 10 milliseconds, as shown in [Figure 2-12](#).

[Figure 2-12. Filter on Duration set to greater than or equal to 10 milliseconds](#)



Once you're finished, click Run to start the trace, then immediately click the Stop button. Because of the workflow required by the SQL Server Profiler user interface, you must actually start a trace before you can script it. On the File menu, select Export, then Script Trace Definition, then For SQL Server 2005. This will produce a script similar to the following (note, we've edited SQL Server Profiler's output a bit for brevity and readability):

Code View:

```

declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

exec @rc = sp_trace_create
    @TraceID output,
    0,
    N'InsertFileNameHere',
    @maxfilesize,
    NULL

if (@rc != 0) goto finish
-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on
-- Set the Filters
declare @bigintfilter bigint
set @bigintfilter = 10000
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter
-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1
-- display trace id for future references
select TraceID=@TraceID
finish:
go

```

Note



An option also exists to script the trace definition for SQL Server 2000. The SQL Trace stored procedures did not change much between SQL Server 2000 and SQL Server 2005, but several new events and columns were added to the product. Scripting for SQL Server 2000 simply drops from the script any events that are not backward-compatible.

This script is an extremely simple yet complete definition of a trace that uses the file provider. A couple of placeholder values need to be modified, but for the most part this is totally functional as-is. Given the complexity of working directly with the SQL Trace stored procedures, we generally define a trace using SQL Server Profiler's UI, then script it and work from there. This way you get the best of both worlds: ease of use combined with the efficiency of server-side traces using the file provider.

This script does a few different things, so we will walk through each stage:

1. First, the script defines a few variables to be used in the process. The @rc variable will be used to get a return code from `sp_trace_create`. The @TraceID variable will hold the handle to the newly created trace. Finally, the @maxfilesize variable defines the maximum size (in MB) per trace file. When running server-side traces, the file provider can be configured to automatically create rollover files as the primary trace file fills up. This can be useful if you're working on a drive with limited space, as you can move previously filled files off to another device. In addition, smaller files can make it easier to manipulate subsets of the collected data. Finally, rollover files also have their utility in high-load scenarios, as will be discussed later in this chapter. However, most of the time these are not necessary, and a value of 5 is a bit small for the majority of scenarios. We'll further discuss this value, as well as uses for rollover files, in the "[Tracing Considerations and Design](#)" section.
2. Next, the script calls the `sp_trace_create` stored procedure, which initializesâ but does not startâ the trace. The parameters specified here are the output parameter for the trace ID of the newly created trace; 0 for the options parameterâ meaning that rollover files should not be used; a placeholder for a server-side file path, which should be changed before using this script; the maximum file size as defined by the @maxfilesize variable; and NULL for the stop dateâ this trace will only stop when it is told to. Note that there is also a final parameter into `sp_trace_create`, which allows the user to set the maximum number of rollover files. This parameter, called @filecount in the `sp_trace_create` documentation, is new in SQL Server 2005 and is not added automatically to the trace definition scripts created with the Script Trace Definition option. The @filecount parameter doesn't apply here, since the options parameter was set to 0 and no rollover files will be created, but it can be useful in many other cases. Note that since rollover files are disabled, if the maximum file size is reached the trace will automatically be stopped and closed.

Note



The file extension .trc will automatically be appended to the file path specified for the output trace file. If you use the .trc extension in your file name, for example C:\mytrace.trc, the file on disk will be C:\mytrace.trc.trc.

3. Next, `sp_trace_setevent` is used to define the event/column combinations used for the trace. In this case, to keep things simple, only event 12â SQL:BatchCompletedâ is used. One call to `sp_trace_setevent` is required for each event/column combination used in the trace. As an aside, note that the @on parameter must be a bit. Since numeric literals in SQL Server are implicitly cast as integers by default, the local @on variable is needed to force the value to be treated appropriately by the stored procedure.
4. Once events are set, filters are defined. In this case, column 13 (Duration) is filtered using the "and" logical operator (the third parameter, with a value of 0) and the "greater than or equal to" comparison operator (value of 4 in the fourth parameter). The actual value is passed in as the final parameter. Note that it is shown in the script in microseconds; SQL Trace uses microseconds for its durations, although the default standard of time in SQL Server Profiler is milliseconds. To change the SQL Server Profiler default, click Tools, then Options, and check the box for Show values in Duration column in microseconds (SQL Server 2005 only).

Note



SQL Trace offers both "and" and "or" logical operators that can be combined if multiple filters are used. However, there is no way to indicate parentheses or other grouping constructs, meaning that the order of operations is limited to left-to-right evaluation. This means that an expression such as A and B or C and D is logically evaluated by SQL Trace as (((A and B) or C) and D). However, SQL Trace will internally break the filters into groups based on columns being filtered. So the expression Column1=10 or Column1=20 and Column3=15 or Column3=25 will actually be evaluated as (Column1=10 or Column1=20) and (Column3=15 or Column3=25). Not only is this somewhat confusing, but it can make certain conditions difficult or impossible to express. Keep in mind that in some cases you may have to break up your filter criteria and create multiple traces in order to capture everything the way you intend to intended.

5. The trace has now been created, event and column combinations set, and filters defined. The final thing to do is actually start tracing. This is done via the call to `sp_trace_setstatus`, with a value of 1 for the second parameter.

Querying Server-Side Trace Metadata

After modifying the file name placeholder appropriately and running the test script on my server, I got back a value of 2 for the trace ID. Using a trace ID you can retrieve a variety of metadata about the trace from the `sys.traces` catalog view, such as is done by the following query:

```
SELECT
    status,
    path,
    max_size,
    buffer_count,
    buffer_size,
    event_count,
    dropped_event_count
FROM sys.traces
WHERE id = 2
```

This query returns the trace status, which will be 1 (started) or 0 (stopped); the server-side path to the trace file (or NULL if the trace is using the rowset provider); the maximum file size (or again, NULL in the case of the rowset provider); information about how many buffers of what size are in use for processing the I/O; the number of events captured; and the number of dropped events (in this case, NULL if your trace is using the file provider).

Note



For readers migrating from SQL Server 2000, note that the `sys.traces` view replaces the older `fn_trace_getinfo` function. This older function returns only a small subset of the data returned by the `sys.traces` view, so it's definitely better to use the view going forward.

In addition to the `sys.traces` catalog view, SQL Server ships with a few other views and functions to help derive other information about traces running on the server.

fn_trace_geteventinfo

This function returns the numeric combinations of events and columns selected for the trace, in a tabular format. The following Transact-SQL returns this data for trace ID 2:

```
SELECT *
FROM fn_trace_geteventinfo(2)
```

The output from running this query on the trace created in the preceding script is shown in [Table 2-1](#).

Table 2-1. Selected Trace Events and Columns as Returned by for a Specific Trace

eventid	columnid	1211261291210121112121312141215121612171218
---------	----------	---

and

The numeric representations of trace events and columns are not especially interesting on their own. In order to be able to properly query this data a textual representation is necessary. The sys.trace_events and sys.trace_columns contain not only text describing the events and columns respectively, but also other information such as data types for the columns and whether they are filterable. Combining these views with the above query against the fn_trace_geteventinfo function, we can get a much easier to read version of the same output:

```
SELECT
    e.name AS Event_Name,
    c.name AS Column_Name
FROM fn_trace_geteventinfo(2) ei
JOIN sys.trace_events e ON ei.eventid = e.trace_event_id
JOIN sys.trace_columns c ON ei.columnid = c.trace_column_id
```

The output from this query is shown in [Table 2-2](#).

Table 2-2. Selected Trace Events and Columns as Rendered To Test for a Specific Trace

Event_Name	Column_Name	SQL:BatchCompleted	TextData	SQL:BatchCompleted	NTUserName	SQL:BatchCompleted
------------	-------------	--------------------	----------	--------------------	------------	--------------------

fn_trace_getfilterinfo

To get information about which filter values were set for a trace, the fn_trace_getfilterinfo function can be used. This function returns the column ID being filtered (which can be joined to the sys.trace_columns view for more information), the logical operator, comparison operator, and the value of the filter. Following is an example of its use:

```
SELECT
    columnid,
    logical_operator,
    comparison_operator,
    value
FROM fn_trace_getfilterinfo(2)
```

Retrieving Data from Server-Side Traces

Once a trace is started, the obvious next move is to actually read the collected data. This is done using the `fn_trace_gettable` function. This function takes two parameters: The name of the first file from which to read the data, and the maximum number of rollover files to read from (should any exist). The following Transact-SQL code reads the trace file located at `C:\inside_sql.trc`:

```
SELECT *
FROM fn_trace_gettable('c:\inside_sql.trc', 1)
```

A trace file can be read at any time, even while a trace is actively writing data to it. Note that this is probably not a great idea in most scenarios, as it will increase disk contention, thereby decreasing the speed with which events can be written to the table and increasing the possibility of blocking. However, in situations in which you're collecting data infrequently—such as when you've filtered for a very specific stored procedure pattern that isn't called often—this is an easy way to find out what you've collected so far.

Because `fn_trace_gettable` is a table-valued function, its uses within Transact-SQL are virtually limitless. It can be used as-is to formulate queries, or inserted into a table so that indexes can be created. In the latter case, it's probably a good idea to use `SELECT INTO`, in order to take advantage of minimal logging:

```
SELECT *
INTO inside_sql_trace
FROM fn_trace_gettable('c:\inside_sql.trc', 1)
```

Once the data has been loaded into a table, it can be sliced and diced any number of ways in order to troubleshoot or answer questions. I'll describe several such queries in the "[Troubleshooting and Analysis with Traces](#)" section later in this chapter.

Stopping and Closing Traces

When a trace is first created, it has the status of 0, stopped (or not yet started, in that case). A trace can be brought back to that state at any time, using `sp_trace_setstatus`. To set trace ID 2 to a status of stopped, the following Transact-SQL code is used:

```
EXEC sp_trace_setstatus 2, 0
```

Aside from the obvious benefit of the trace no longer collecting data, there is another perk to doing this: Once the trace is in a stopped state, you can modify the event/column selections and filters using the appropriate stored procedures, without re-creating the trace. This can be extremely useful if you only need to make a minor adjustment.

If you are actually finished tracing and do not wish to continue at a later time, you can remove the trace definition from the system altogether by setting its status to 2:

```
EXEC sp_trace_setstatus 2, 2
```

Note



Trace definitions will be automatically removed in the case of a SQL Server service restart, so if you do need to run the same trace again later either save it as a Profiler template or save the script used to start it.

Investigating the Rowset Provider

Most of this section has dealt with how to work with the file provider using server-side traces, but some readers are undoubtedly asking themselves how SQL Server Profiler interfaces with the rowset provider. The rowset provider and its interfaces are completely undocumented. However, because SQL Server Profiler is doing nothing more than calling stored procedures under the covers, it is not too difficult to find out what's going on. As a matter of fact, you can use a somewhat recursive process: use SQL Server Profiler to trace activity generated by itself.

A given trace session will not be able to capture all of its own events (the trace won't be running yet when some of them occur), so to see how Profiler works we need to set up two traces: an initial trace configured to watch for SQL Server Profiler activity, and a second trace to produce the activity for the first trace to capture. To begin with, open SQL Server Profiler and create a new trace using the default template. In the Edit Filter dialog box, remove the default Not Like filter on ApplicationName, and replace it with a Like filter on ApplicationName for the string SQL Server Profiler%. This filter will capture all activity that is produced by any SQL Server Profiler session.

Start that trace, then load up another trace using the default template and start it. The first trace window will now fill with calls to the various sp_trace stored procedures, fired via RPC:Completed events. The first hint that something different happens when using the rowset provider is the call made to sp_trace_create:

```
declare @p1 int
exec sp_trace_create @p1 output, 1, NULL, NULL, NULL
select @p1
```

The second parameter, used for options, is set to 1, a value not documented in SQL Server Books Online. This is the value that turns on the rowset provider. And the remainder of the parameters, which deal with file output, are populated with NULLs.

Note



The sp_trace_create @options parameter is actually a bitmask—multiple options can be set simultaneously. To do that, simply add up the values for each of the options you'd like. With only three documented values and one undocumented value there aren't a whole lot of possible combinations, but it's still something to keep in mind.

Much of the rest of the captured activity will look familiar at this point; you'll see normal-looking calls to sp_trace_setevent, sp_trace_setfilter, and sp_trace_setstatus. However, to see the complete picture you'll have to stop the second trace (the one actually generating the trace activity being captured). As soon as the second trace stops, the first trace will capture the following RPC:Completed event:

```
exec sp_executesql N'exec sp_trace_getdata @P1, 0',N'@P1 int',3
```

In this case, 3 is the trace ID for the second trace on our system. Given this set of input parameters, the `sp_trace_getdata` stored procedure streams event data back to the caller in a tabular format, and does not return until the trace is stopped. The second parameter is undocumented and doesn't need to be set.

Unfortunately, the tabular format produced by `sp_trace_getdata` is far from recognizable, and is not in the standard trace table format. By modifying the previous file-based trace, we can produce a rowset-based trace using the following Transact-SQL:

Code View:

```
declare @rc int
declare @TraceID int

exec @rc = sp_trace_create
    @TraceID output,
    1,
    NULL,
    NULL,
    NULL
if (@rc != 0) goto finish
-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on
-- Set the Filters
declare @bigintfilter bigint
set @bigintfilter = 10000
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter
-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1
-- display trace id for future references
select TraceID=@TraceID
exec sp_executesql
    N'exec sp_trace_getdata @P1, 0',
    N'@P1 int',
    @TraceID
finish:
go
```

Running this code, then issuing a `WAITFOR DELAY '00:00:10'` in another window, produces the output (truncated and edited for brevity), as shown in [Table 2-3](#).

Table 2-3. Output from the Rowset Provider as Exposed via for our trace

```
columnidLengthData6552660xFEFF6300000014160xD707050002001D001 . .
.65533310x01010000000300000 . .65532260x0C000100060009000 . .65531140x0D000004080010270 . .
.6552660xFAFF000000006552660x0C000E010001480x57004100490054004 . .
.680x4100640061006D00940xC813000010920x4D006900630072006 . .
```

Each of the values in the columnid column corresponds to a trace data column ID. The length and data columns are relatively self-explanatory—data is a binary-encoded value that corresponds to the collected column, and length is the number of bytes used by the data column. Each row of the output coincides with one column of one event. SQL Server Profiler pulls these events from the rowset provider via a call to sp_trace_getdata and performs a "pivot" to produce the human-readable output we're used to seeing. This is yet another reason that the rowset provider can be less efficient than the file provider—sending so many rows can produce a huge amount of network traffic.

If you do require rowset provider-like behavior for your monitoring needs, you luckily will not need to figure out how to manipulate this data. SQL Server 2005 ships with a series of managed classes in the Microsoft.SqlServer.Management.Trace namespace, designed to help with setting up and consuming rowset traces. The use of these classes is beyond the scope of this chapter, but they are well documented in the SQL Server TechCenter on TechNet, and readers should have no trouble figuring out how to exploit what they offer.

Now that we've explored some of the ins and outs of SQL Trace and SQL Server Profiler, the next section will take things to a slightly more practical level, showing you how you can apply SQL Trace to a variety of real-world problems.

Troubleshooting and Analysis with Traces

The goal of SQL Trace is to help you—presumably, a DBA or database developer—analyze your system in order to either detect or solve problems. As SQL Server is such a large and complex product, using SQL Trace without knowing exactly what you want to do can sometimes feel like trying to find the proverbial needle in a very large haystack. In this section, we'll try to distill some of the lessons we've learned from doing countless traces into a solid framework upon which you can build your own tracing skill. I'll start with an overview of those event classes that we use most often, after which we'll drill into some common scenarios for which SQL Trace can help in your day-to-day work.

Commonly Used SQL Trace Event Classes

With over 170 events to choose from, the Profiler Events Selection dialog can be a bit daunting to new users. Luckily for your sanity, most of these events are not commonly used in day-to-day tracing activity, and are provided more for the sake of automated tools. In this section, we will describe the most common events that we use in our work.

- Errors and Warnings:Attention

This event is usually fired when a client disconnects from SQL Server unexpectedly. The most common cause of this is a client library time-out, which is generally a 30-second timer that starts the moment a query is submitted. If queries are timing out, it's something you want to know about immediately, so this event is used frequently.

- Errors and Warnings:Exception and Errors and Warnings:User Error Message

Exceptions and user error messages go hand-in-hand, and we always trace these two classes together.

When a user exception occurs, both events will be fired. The Exception event will include the error number, severity, and state, whereas the User Error Message event will include the actual text of the error. We discuss these events in detail in the section "[Identifying Exceptions](#)" later in this chapter.

- Locks:Deadlock Graph and Locks:Lock:Deadlock Chain

In previous versions of SQL Server, deadlocks could only be identified through the slightly obscure Deadlock Chain event. SQL Server 2005 introduces the much more usable Deadlock Graph event, which produces standard XML that Profiler is able to render into a very nice graphical output. We'll discuss this output in the "[Debugging Deadlocks](#)" section later in this chapter.

- Locks:Lock:Acquired, Locks:Lock:Released, and Locks:Lock:Escalation

We use these events mainly in conjunction with working on deadlocks. They provide insight into what locks SQL Server takes during the course of a transaction, and for how long they are held. These events can be also very interesting to monitor if you're curious about how SQL Server's various isolation levels behave. Make sure, when using these events, to filter on a specific spid that you're targeting, lest you get back far too much information to process.

- Performance>Showplan XML Statistics Profile

This event can be used to capture XML showplan output for queries that have run on the server you're profiling. There are actually several different showplan and XML showplan event classes, but this one is the most useful, in our opinion, as it includes actual rowcounts and other statistics data, which can help when tuning queries. This event class will be discussed in more detail in the "[Performance Tuning](#)" section.

- Security Audit (Event Category)

Although this isn't an event classâ€“ but rather, a category that includes many event classesâ€“ we thought it belonged in this list because it contains a number of useful event classes to help you monitor virtually all security-related activity as it occurs on your server. This includes such information as failed logins attempts (Audit Login Failed event class), access to specific tables or other objects (Audit Schema Object Access Event event class), and even when the server is restarted (Audit Server Starts and Stops event class). Most of these event classes are designed for SQL Server's built-in server auditing traces, described in the "[Auditing: SQL Server's Built-in Traces](#)" section at the end of this chapter.

- Security Audit:Audit Login and Security Audit:Audit Logout

We specifically singled these two events out of the overall Security Audit category because they are the two events in that category that we find useful on a day-to-day basis, especially when doing performance tuning. By monitoring these events along with various query events in the Stored Procedures and TSQL category, you can more easily aggregate based on a single session.

Tip



Thanks to a change in SQL Server 2005 SP2, these events now fire even for pooled logins and logouts, making them even more useful than before. To detect whether or not the event fired based on a pooled connection, look for a value of 2 in the EventSubClass column.

- Stored Procedures:RPC:Starting and Stored Procedures:RPC:Completed

These events fire when a remote procedure call (RPC; generally, a parameterized query or stored procedure call, depending on the connection library you're using) is executed by a client application.

- TSQL:SQL:BatchStarting and TSQL:SQL:BatchCompleted

These events fire when an ad hoc batch is executed by a client application. Using these in combination with the RPC event classes will allow you to capture all requests submitted to the server by external callers. Both the BatchCompleted event class and the corresponding RPC:Completed event class populate four key columns: Duration, Reads, Writes, and CPU. We discuss these in more detail in the "Performance Tuning" section later in this chapter.

- Stored Procedures:SP:StmtStarting and Stored Procedures:SP:StmtCompleted

Sometimes it can be difficult to determine which access path was taken in a complex stored procedure full of flow control statements. These events are fired every time a statement in a stored procedure is executed, giving you a full picture of what took place. Note that these events can produce an extremely large amount of data. It is, therefore, best to use them only when you've filtered the trace by either a given spid you're tracking, or a certain stored procedure's name or object ID (using the ObjectName or ObjectId columns, respectively) that you're interested in.

- Stored Procedures:SP:Recompile

Stored procedure recompiles are commonly cited as a potential SQL Server performance problem. SQL Server includes a counter to help track them (SQLServer:SQL Statistics:SQL Re-Compilations/Sec), and if you see a consistently high value for this counter you may consider profiling using this event class in order to determine which stored procedures are causing the problem.

Note



For more information on avoiding recompiles see [Chapter 5, "Plan Caching and Recompilation."](#) In addition, Microsoft has published a very good knowledge base article on the topic, which you can access at the following URL:
<http://support.microsoft.com/default.aspx?scid=kb;en-us;Q243586>.

- Stored Procedures:SP:Starting

This event class fires whenever a stored procedure or function is called—no matter whether it is called directly by a client, or nested within another stored procedure or function. Since it does not populate the Reads, Writes, or CPU columns, this event class is not especially useful for performance tuning, but it does have its value. We use this class often for obtaining simple counts of the number of times a specific stored procedure was called within a given interval, and also for situations in which stored procedure calls are heavily nested and we need to determine exactly in which sequence calls were made that resulted in a certain procedure getting executed.

- Transactions:SQLTransaction

This event can be used to monitor transaction starts, commits, and rollbacks. You can determine which state the transaction is in by looking at the EventSubClass column, which will have a value of 0, 1, or 2 for a transaction starting, committing, or rolling back, respectively. Note that because every data modification uses a transaction, this event can cause a huge amount of data to be returned on a busy server. If possible, make sure to filter your trace based on a specific spid that you're tracking.

- User Configurable (Event Category)

This event category contains 10 events, named UserConfigurable:0 through UserConfigurable:9. These events can be fired by users or modules with sufficient ALTER TRACE access rights, allowing custom data to be traced. We discuss some possibilities in the "Stored Procedure Debugging" section of this chapter.

Performance Tuning

Performance is always a hot topic, and for a good reason; in today's competitive business landscape, if your users feel that your application is too slow they'll simply move on to a different provider. In order to help you prevent this from happening, SQL Trace comes loaded with several event classes that you can harness in order to find and debug performance bottlenecks.

Note



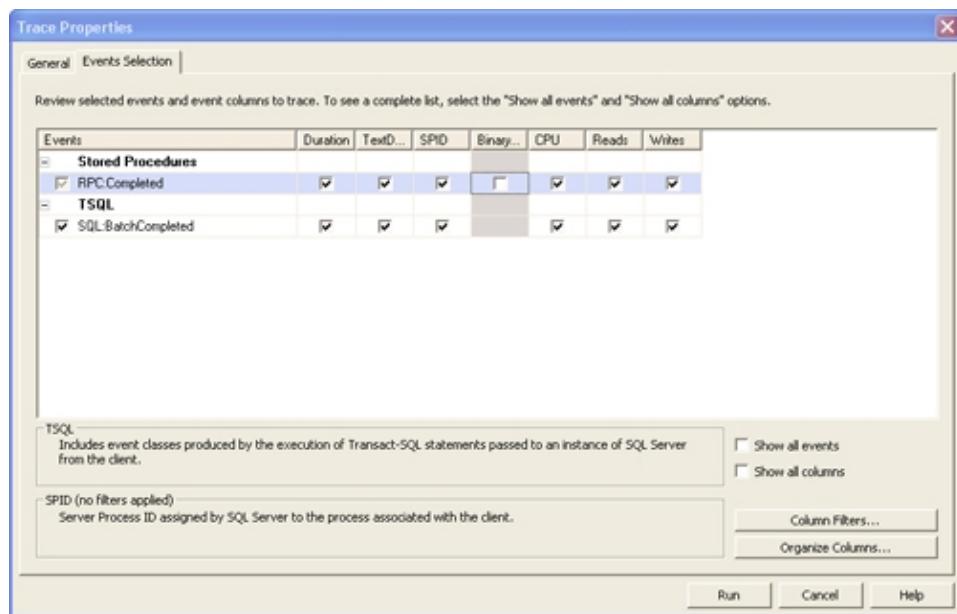
Performance monitoring is an art unto itself, and a complete methodology is outside the scope of this chapter; in this chapter, we'll stick with only what you can do using SQL Trace. Different aspects of troubleshooting for performance are discussed throughout this book. In addition, for a comprehensive discussion on the topic of how to think about SQL Server performance, please refer to *Inside SQL Server 2005: T-SQL Querying* (Microsoft Press, 2006).

Performance monitoring techniques can be roughly grouped into two categories: those you use when you already know something about the problem, and those you use to find out what the problem is (or find out if there even is a problem). These categories tend to dovetail nicely; once you've found out something about the problem, you can start heading in the direction of getting more information. Therefore, let's start with the second technique, helping to pinpoint problem areas, and then move into how to conduct a more detailed analysis.

When walking into a new database performance tuning project, the very first thing we want to find out is what queries make up the "low-hanging fruit." In other words, we want to identify the worst performance offenders, as these are the ones that can give us the biggest tuning gains. It's important at this stage to not trace too much information, so we generally start with only the `Stored Procedures:RPC:Completed` and `TSQL:SQL:BatchCompleted` events. These events are both selected in the `TSQL_Duration` template that ships with SQL Server Profiler. We recommend adding the `Reads`, `Writes`, and `CPU` columns for both events, which are not selected by default in the template, in order to get a more complete picture. We also recommend selecting the `TextData` column rather than the (default) `BinaryData` column for the `RPC:Completed` eventâ€”this will make it easier to work with the data later. A properly specified set of events is shown in [Figure 2-13](#).

Figure 2-13. Selected events/columns for an initial performance audit

[\[View full size image\]](#)



Once you've selected the events, set a filter on the Duration column for a small number of milliseconds. Most of the active OLTP systems we've worked with have an extremely large number of zero-millisecond queries, and these are clearly not the worst offenders in terms of easy-to-fix performance bottlenecks. We generally start with the filter set to 100 milliseconds, and work our way up from there. The idea is to increase the signal-to-noise ratio on each iteration, eliminating "smaller" queries and keeping only those that have a high potential for performance tuning. Depending on the application, we generally run each iterative trace for 10 to 15 minutes, depending on server load, then take a look at the results and increase the number appropriately until we net only a few hundred events over the course of the trace. Note that the 10- to 15-minute figure may be too high for some extremely busy applications. See the "["Traceing Considerations and Design"](#)" section later in this chapter for more information.

The other option is to run only the initial trace, and then filter the results down from there. A simple way to handle that is with SQL Server 2005's NTILE windowing function, which divides the input rows into an equal number of "buckets." To see only the top 10 percent of queries in a trace table, based on duration, use the following query:

```
SELECT *
FROM
(
    SELECT
        *,
        NTILE(10) OVER (ORDER BY Duration) Bucket
    FROM TraceTable
) x
WHERE Bucket = 10
```

Note



The execution by an application of an extremely large number of seemingly smallâ even zero-millisecondâ queries can also be a performance problem, but it generally has to be solved architecturally, by removing chatty interfaces, rather than via Transact-SQL query tuning. Profiling to find these kinds of issues can also be incredibly difficult without knowledge of exactly what a particular application is doing, so we will

not cover it here.

If you find that it is difficult to get the number of events returned to a manageable levelâ a common problem with very busy systemsâ you may have to do some tweaking of the results to group the output a bit better. The results you get back from SQL Trace will include raw text data for each query, including whichever actual arguments were used. To analyze the results further, the data should be loaded into a table in a database, then aggregated, for example, to find average duration or number of logical reads.

The problem is doing this aggregation successfully on the raw text data returned by the SQL Trace results. The actual arguments are good to knowâ they're useful for reproducing performance issuesâ but when trying to figure out which queries should be tackled first we find that it's better to aggregate the results by query "form." For example, the following two queries are of the same formâ they use the same table and columns, and only differ in the argument used for the WHERE clauseâ but because their text is different it would be impossible to group them as-is:

```
SELECT *
FROM SomeTable
WHERE SomeColumn = 1
---
SELECT *
FROM SomeTable
WHERE SomeColumn = 2
```

To help solve this problem, and reduce these queries to a common form that can be grouped, Itzik Ben-Gan provided a CLR UDF in Inside SQL Server 2005: T-SQL Querying, a slightly modified version of whichâ that also handles NULLsâ follows:

```
[Microsoft.SqlServer.Server.SqlFunction(IsDeterministic=true)]
public static SqlString sqlsig(SqlString querystring)
{
    return (SqlString)Regex.Replace(
        querystring.Value,
        @"([\s,(<>!) (?![^\]]]+[\]])|(?:(?:((?:(?:(?# expression coming
            )|([N])?(')(?:[^']|'')*('))|(?#
            )|(?:(0x[\da-fA-F]*))|(?# character
            )|(?:(-+)?(?:(\d)*\.\d*\|\d+))|(?# binary
            )|(?:(eE)?[\d*]))|(?# precise number
            )|(?:[~]?[-+]?(?:[\d]+))|(?# imprecise number
            )|(?:[nN][uU][lL][lL])|(?# integer
            )|(?:[\s]?[+\-\/*\%\&\|^][\s]?)|(?# null
            ))|(?:(?:[\s]?[+\-\/*\%\&\|^][\s]?)?)|(?# operators
            ))",
        @"$1$2$3#$4");
}
```

This UDF finds most values that "look" like arguments, replacing them with a "#". After processing both of the preceding queries with the UDF, the output would be the same:

```
SELECT *
FROM SomeTable
WHERE SomeColumn = #
```

To use this UDF to help with processing a trace table to find the top queries, you might start with something along the lines of the following query, which groups each common query form and finds average values for Duration, Reads, Writes, and CPU:

```

SELECT
    QueryForm,
    AVG(Duration),
    AVG(Reads),
    AVG(Writes),
    AVG(CPU)
FROM
(
    SELECT
        dbo.fn_sqlsig(TextData) AS QueryForm,
        1.* Duration AS Duration,
        1.* Reads AS Reads,
        1.* Writes AS Writes,
        1.* CPU AS CPU
    FROM TraceTable
    WHERE TextData IS NOT NULL
) x
GROUP BY QueryForm

```

From here you can further filter by the average values in order to find those queries to which you'd like to dedicate a bit more attention.

Once you have decided upon one or more queries to tune, you can use SQL Trace to help with further analysis. For example, suppose that you had isolated the following stored procedure, which can be created in the AdventureWorks database, as a culprit:

```

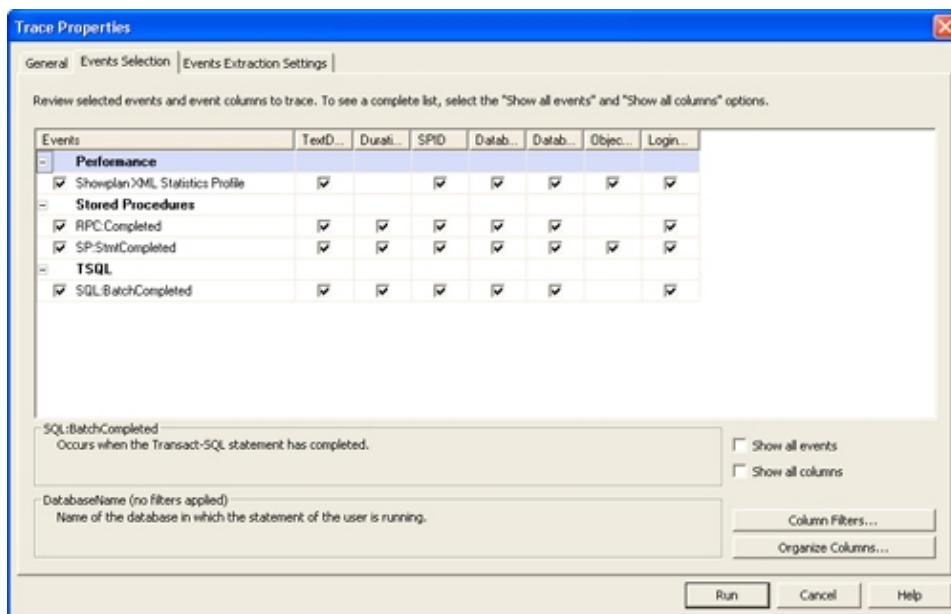
CREATE PROCEDURE GetManagersAndEmployees
    @EmployeeID INT
AS
BEGIN
    SET NOCOUNT ON
    EXEC uspGetEmployeeManagers @EmployeeID
    EXEC uspGetManagerEmployees @EmployeeID
END

```

To begin a session to analyze what this stored procedure is doing, first open a new query window in SQL Server Management Studio, and get the spid of your session using the @@SPID function. Next, open SQL Server Profiler, connect to your server, and select the Tuning template. This template adds SP:StmtCompleted to the combination of events used to get a more general picture of server activity. This will result in a lot more data returned per call, so use the spid you collected before to filter your trace. You also might wish to add the Showplan XML Statistics Profile event, in order to pull back query plans along with the rest of the information about your query. [Figure 2-14](#) shows a completed Events Selection screen for this kind of work.

Figure 2-14. Trace set up for performance profiling of a single spid's workload

[\[View full size image\]](#)



Note

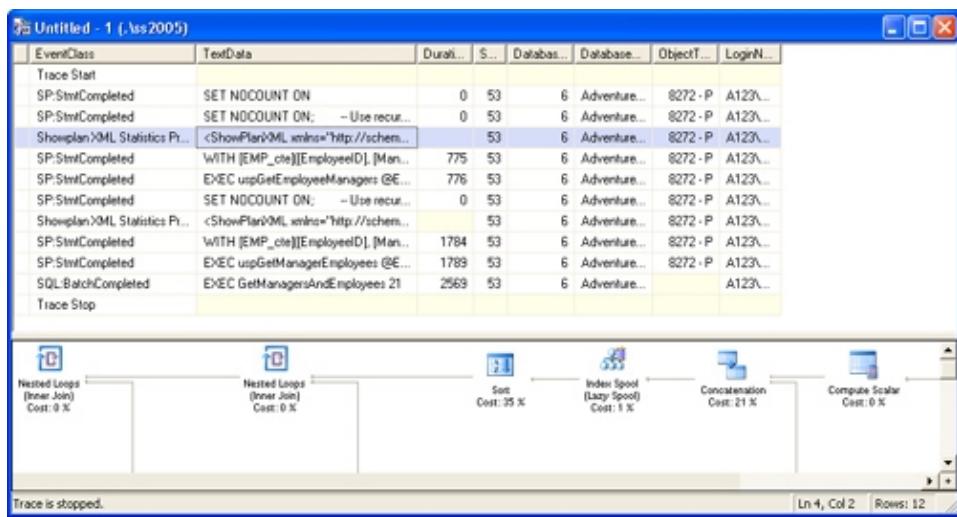


Adding a Showplan XML or Deadlock Graph event causes an additional tab to open in the Trace Properties dialog, called Events Extraction Settings. This tab contains options for automatically saving any collected query plans or deadlock graph XML to text files, in case you need to reuse them later.

Next, go ahead and start the trace in SQL Server Profiler. Although we generally use server-side traces for most performance monitoring, the amount of overhead that Profiler brings to the table when working with a single spid to process a single query is small enough that we don't mind harnessing the power of the UI for this kind of work. [Figure 2-15](#) shows the Profiler output on our end after starting the trace and running the query for @EmployeeID=21. We have selected one of the Showplan XML events in order to highlight the power of this feature; along with each statement executed by the outermost stored procedure and any stored procedures it calls, you can see a full graphical query plan, all in the Profiler UI. This makes it an ideal assistant for helping you tune complex, multilayered stored procedures.

Figure 2-15. Using Profiler to trace batches, statements, and query plans

[[View full size image](#)]



SQL Trace will not actually tune for you, but it will help you to find out not only which queries are likely to be causing problems, but also which components of those queries need work. However, performance tuning is far from the only thing that it can be used for. In the next section, we'll explore another problem area—exceptions—and how to use SQL Trace to help in tracking them down.

Note



Saying that SQL Trace will not actually tune for you is only mostly correct. SQL Server's Database Engine Tuning Advisor (DTA) tool can take trace files as an input, in order to help recommend indexes, statistics, and partitions to make your queries run more quickly. If you do use the DTA tool, make sure to feed it with a large enough sample of the queries your system generally handles. Too small a collection size will skew the results, potentially causing the DTA to make subpar recommendations, or even produce suggestions that cause performance problems for other queries that weren't in the input set.

Identifying Exceptions

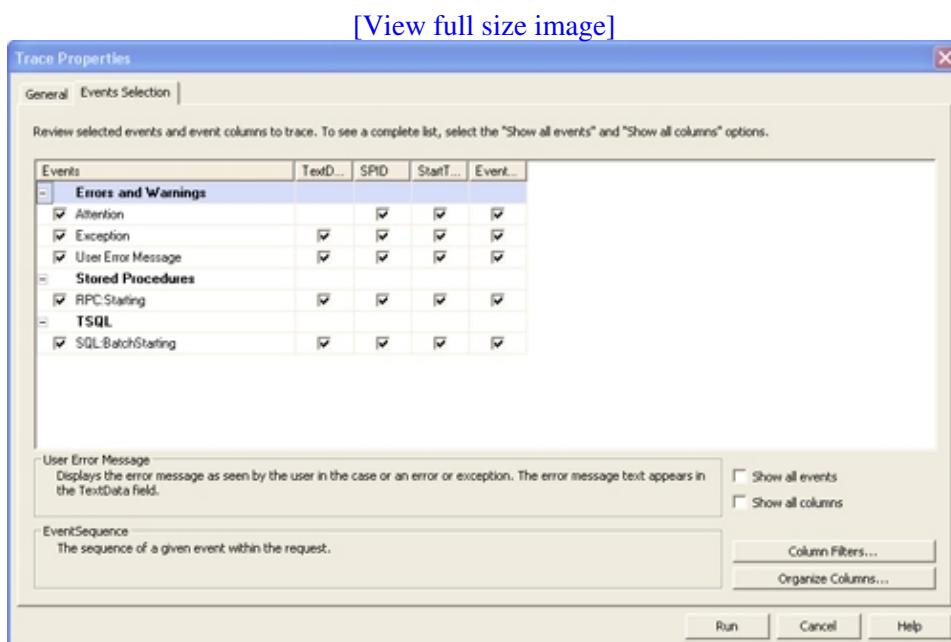
In a perfect world, all exceptions would be caught, handled, and logged. Appropriate personnel would regularly watch the logs and create bug reports based on the exceptions that occurred, such that they could be debugged in a timely manner and avoided in the future. But alas, the world is far from perfect, and it is quite common to see applications that regularly bubble exceptions all the way from the database to the user interface, with no logging of any kind. Or, worse, applications that catch and swallow exceptions, resulting in no one ever knowing that they occurred, yet occasionally resulting in strange data returned to users who aren't quite sure what's going on. In order to find out when either of these scenarios is happening, we often take a proactive approach and watch for exceptions so that we can find and fix them—hopefully before they frustrate too many users.

Tracing for exceptions is fairly simple; you can start with the TSQL template, which includes the Audit Login and Audit Logout events, as well as the ExistingConnection event—all of which can be removed for this exercise. Left are RPC:Starting and SQL:BatchStarting, both of which are needed in order to trace the Transact-SQL that caused an exception, whether it occurred as the result of an SQL batch or RPC call. It's important to trace for the Starting rather than Completed events in this case, because certain errors can result in the Completed event not firing for a given query.

To the RPC and SQL events, add the Attention, Exception, and User Error Message events from the Errors and Warnings category. The Attention event fires whenever a client forcibly disconnects—the best example of which is a client-side query time-out. These are good to know about, and usually indicate performance or blocking issues. The Exception event fires whenever an exception of any kind occurs, whereas the User Error Message event fires either in conjunction with an exception—to send back additional data about what occurred, in the form of a message—or whenever a variety of statuses change, such as a user switching from one database to another.

We also recommend adding the EventSequence column for each selected event class. This will make querying the data later much easier. [Figure 2-16](#) shows a completed Events Selection dialog box for the events and columns that we recommend for exception monitoring.

Figure 2-16. Events Selection used for tracing to discover exceptions and disconnections



Note



SQL Server uses exceptions internally to send itself messages during various phases of the query execution process. A telltale sign of this is an Exception event with no corresponding User Error Message event. Should you see such a situation, do not be alarmed; it's not actually an error meant for you to consume.

Once you've selected the appropriate events, script the trace and start it. This is the kind of trace you may want to run in the background for a while, doing occasional collections. See the "[Tracing Considerations and Design](#)" section for more information. Generally, you may find it interesting to collect this data during fairly busy periods of activity, in order to find out what exceptions your users may be experiencing. Since this kind of tracing is more like casting a net and hoping to catch something than a hunt for something specific, timing is of the essence. You may or may not see an exception, but just because you don't see one during one collection period doesn't mean they aren't happening; make sure to monitor often.

Once you've captured your data and transferred it into a table, reporting on which exceptions occurred is a matter of finding:

- All Attention events (EventClass 16) and the Transact-SQL or RPC event (EventClass 13 and 10, respectively) on the same spid that preceded the disconnection.
- All Exception events (EventClass 33) immediately followed by a User Error Message event (EventClass 162), and the Transact-SQL or RPC event on the same spid that preceded the exception.

All logic for following and preceding can be coded using the EventSequence column, which is why we recommend including it in this trace. The following query uses this logic to find all user exceptions and disconnections, related error messages where appropriate, and the queries that caused the problems:

Code View:

```
;WITH Exceptions AS
(
    SELECT
        T0.SPID,
        T0.EventSequence,
        COALESCE(T0.TextData, 'Attention') AS Exception,
        T1.TextData AS MessageText
    FROM TraceTable T0
    LEFT OUTER JOIN TraceTable T1 ON
        T1.EventSequence = T0.EventSequence + 1
        AND T1.EventClass = 162
    WHERE
        T0.EventClass IN (16, 33)
        AND (T0.EventClass = 16 OR T1.EventSequence IS NOT NULL)
)
SELECT *
FROM Exceptions
CROSS APPLY
(
    SELECT TOP(1)
        TextData AS QueryText
    FROM TraceTable Queries
    WHERE
        Queries.SPID = Exceptions.SPID
        AND Queries.EventSequence < Exceptions.EventSequence
        AND Queries.EventClass IN (10, 13)
        ORDER BY EventSequence DESC
) p
```

If you've collected a large number of events, you can greatly improve the performance of this query by creating a clustered index on the trace table, on the EventSequence column.

Tip



If you're aware of an exception condition that occurs when a certain stored procedure is called, but you need more information on what sequence of events causes it to fire, you might work with the same events detailed in the preceding section, for performance tuning of a single query. Switch the query events shown in that section from Completed to the corresponding Starting classes, and add the Exception and User Error Message events. Much like with the tuning of a single query example, this is something you should run directly in SQL Server Profiler, with a filter on the spid from which you're working.

Debugging Deadlocks

Tracking general exceptions is a good thing, but every DBA knows the horrors of dealing with a certain type exception: message number 1205, severity 13, familiarly known as deadlocks. Deadlock conditions are difficult to deal with because it often feels like you just don't get enough data from the server to help you figure out exactly what happened and why. Even the error message returned by the server is somewhat hopeless; the only suggestion made in the message is that you might "rerun the transaction."

SQL Trace has long exposed tools to help with isolating and debugging deadlock conditions, but SQL Server 2005 takes these to the next level, providing a very useful graphical interface to help you resolve these nasty issues. To illustrate what is available, we'll show you how to force a deadlock in the tempdb database. Start with the following code:

```
USE tempdb
GO
CREATE TABLE Deadlock_Table
(
    ColumnA int NOT NULL PRIMARY KEY
)
GO
INSERT Deadlock_Table
SELECT 1 UNION ALL SELECT 2
GO
```

By starting two separate transactions and staggering updates to the rows in the opposite order, we can make a deadlock occur and observe how SQL Trace can help with debugging.

Note



The following example assumes that you have already identified the two stored procedures or queries involved in causing the deadlock, either based on an exception trace as shown in the preceding section, or by enabling trace flag 1222 at the server level (add `-T1222` to the startup parameters for the SQL Server service) and collecting the deadlocking resources from the SQL Server error log. Once you have identified the participating queries, conduct the actual research on a development instance of SQL Server on which you've restored the production data. Debugging deadlocks can require collecting a lot of data, and because some of the events are fired by system spids it can be impossible to filter the trace so that you only collect relevant data. On a busy system this trace will create a very large amount of load, so we recommend always working offline.

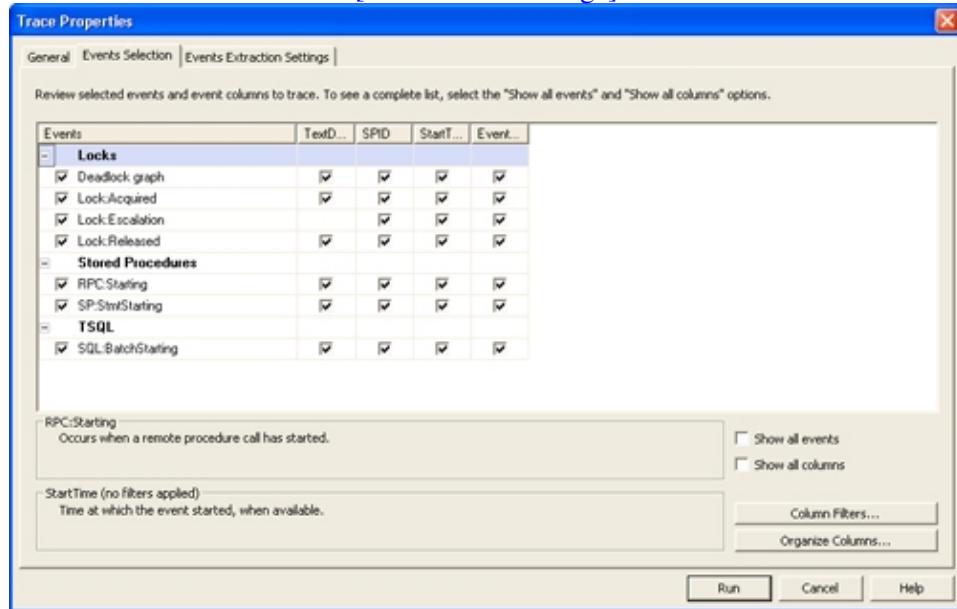
To begin with, open two query windows in SQL Server Management Studio, and collect their spids using `@@SPID`. You may want to use these spids later to help with analysis of the collected trace data. Start a new SQL Server Profiler session and use the TSQL template to select the `RPC:Starting` and `SQL:BatchStarting` events. To these, add the `Deadlock` graph, `Lock:Acquired`, `Lock:Escalation`, and `Lock:Released` events, all from the `Locks` category. The `Lock` events will help you analyze the sequence of locks taken to contribute to the deadlock condition, and the `Deadlock` graph event will help with a graphical display of what went wrong.

You might optionally consider adding the SP:StmtStarting event, in case one or more of the stored procedures you're debugging are running numerous statements, any of which might be contributing to the deadlock. You should also add the EventSequence column, in order to make it easier to analyze the data after collection.

Figure 2-17 shows the completed Events Selection dialog box for this activity.

Figure 2-17. Completed Events Selection dialog box for debugging deadlocking queries

[View full size image]



Tip



In this example, we will show you how to run the statements in exactly the correct order and at the correct times to force a deadlock to occur while you watch the trace in SQL Server Profiler. However, in many real-world situations it's not quite that easy; many deadlocks depend upon exact timing (or mistiming) and to reproduce them you'll have to run each query in a loop, hoping that they eventually collide in just the right way. SQL Server Management Studio also has a feature to help you run a query in a loop as many times as you'd like. In each query window, set up the batch for your queries, then follow the batch with GO and the number of times you'd like the query to run. For example, the following Transact-SQL code will execute the MyStoredProcedure stored procedure 1,000 times:

```
EXEC MyStoredProcedure
```

```
GO 1000
```

Once your events are set up, start the trace in Profiler. Note that no filters should be used in this case, as the Deadlock graph event may be fired by any number of system spids. As a result of the lack of filters, you'll see some system lock activity. This can be ignored or filtered out after the trace has been completed. Another strategy is covered later in this chapter, in the "Reducing Trace Overhead" section.

Once the trace is started, return to the first query window (spid 52 in our test), and run the following batch:

```
BEGIN TRANSACTION
UPDATE Deadlock_Table
SET ColumnA = 3
WHERE ColumnA = 1
GO
```

Next, run the following batch in the second query window (spid 53 in our test):

```
BEGIN TRANSACTION
UPDATE Deadlock_Table
SET ColumnA = 4
WHERE ColumnA = 2
GO
```

Both of these queries should return, since their locks are compatible; they're each taking locks on different rows of the Deadlock_Table table. Back in the first query window, start the following update, which will begin waiting on the second window's session to release its lock:

```
UPDATE Deadlock_Table
SET ColumnA = 4
WHERE ColumnA = 2
GO
```

Finally, return to the second window and run the following update, which will start waiting on the first window's session to release its lock. Since both sessions will now be waiting for each other to release resources, a deadlock will occur:

```
UPDATE Deadlock_Table
SET ColumnA = 3
WHERE ColumnA = 1
GO
```

After the deadlock has occurred the trace can be stopped. Find the Deadlock graph event produced, which should look something like the one shown in [Figure 2-18](#). The Deadlock graph event includes a huge amount of data to help you debug what occurred. Included are the object ID, the index name (if appropriate), and the HoBt (Hash or B-Tree) ID, which can be used to filter the locking resource even further, by using the hobt_id column of the sys.partitions view. In addition, you can determine the actual queries that were involved in the deadlock by scrolling back and finding the last query event run by each spid prior to the deadlock occurring.

Figure 2-18. Viewing the event in SQL Server Profiler

[\[View full size image\]](#)

EventClass	TextData	S...	StartTime	EventSeq...
Lock:Released	{0000c841d5ca}	16	6/7/2007 1:39:48 PM	3804
Lock:Acquired		16	6/7/2007 1:39:48 PM	3805
Lock:Acquired	object_id = 181575605, index_id or st...	16	6/7/2007 1:39:48 PM	3806
Lock:Released	object_id = 181575605, index_id or st...	16	6/7/2007 1:39:48 PM	3807
Lock:Released		16	6/7/2007 1:39:48 PM	3808
Deadlock graph	<deadlock-list> <deadlock victim="p...">	16	6/7/2007 1:39:48 PM	3809
Lock:Acquired		25	6/7/2007 1:39:49 PM	3810
Lock:Acquired		25	6/7/2007 1:39:49 PM	3811
Lock:Acquired		25	6/7/2007 1:39:49 PM	3812
Lock:Acquired	object_id = 2093250512, index_id or ...	25	6/7/2007 1:39:49 PM	3813
Lock:Acquired	schema_id = 4	25	6/7/2007 1:39:49 PM	3814
Lock:Acquired		25	6/7/2007 1:39:49 PM	3815
Lock:Released		25	6/7/2007 1:39:49 PM	3816

Trace is stopped.

Ln 875, Col 1 Rows: 1269

Should you need more data to debug further, you also have a wealth of lock information available. You might notice that in the screenshot, none of the lock events adjacent to the Deadlock graph event have anything to do with the spids with which we were working. The system acquires and releases quite a few locks even while at rest, so to look at the lock chain in more detail you'll want to load the data into a trace table and make use of the EventSequence column to rebuild exactly what happened, in the correct order.

As with performance analysis, SQL Trace will not actually resolve the deadlock condition for you, but it will provide you with ample data to help you determine its cause and get closer to a resolution. In the next section, we'll explore other debugging tricks with SQL Trace.

Stored Procedure Debugging

SQL Server 2005 includes many different tools to help you debug complex stored procedures. These include a Transact-SQL debugger available in Visual Studio, the ability to embed print statements or otherwise return debug statuses from your stored procedures, and the ability to raise custom errors from within your stored procedures in order to log status information. SQL Trace also supplies an underpublicized tool that can be helpful in this regard, called user-configurable events.

A user-configurable event is nothing more than a call to a system stored procedure called `sp_trace_generateevent`. This stored procedure accepts three parameters:

- @eventid is an integer value between 82 and 91. Each value corresponds to one of the 10 user configurable event classes, numbered from 0 through 9; a value of 82 will raise a UserConfigurable:0 event, 83 will raise a UserConfigurable:1 event, etc.
- @userinfo is an nvarchar(128) value that will be used to populate the TextData column for the event.
- @userdata is a varbinary(8000) value that will be used to populate the BinaryData column for the event.

In a few especially tricky situations, we've had to deal with stored procedures that only occasionally failed, under circumstances that were difficult to replicate in a test environment. Long-term tracing in these situations can be difficult, as to really debug the situation might require statement-level collection, which will produce a lot of data if run for extended periods. A better option is to trace at the stored-procedure- and batch-level only, and use user configurable events to collect variable values and other data to help you debug the problem when it does actually occur. This way you can leave the system running and functional, without having to worry about your trace collecting too much data.

To set this up, you must first be aware that the `sp_trace_generateevent` stored procedure requires `ALTER TRACE` permission in order to be run. Since it's unlikely that your stored procedures will have access to that permission, it is a good idea to create a wrapper stored procedure that calls `sp_trace_generateevent`, and which has the appropriate permission. To make this happen, we'll have to employ SQL Server 2005's module signing feature. The first step is to create a certificate in the master database:

```
USE master
GO
CREATE CERTIFICATE ALTER_TRACE_CERT
ENCRYPTION BY PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~'
WITH
    SUBJECT='Certificate for ALTER TRACE',
    START_DATE='20000101',
    EXPIRY_DATE='99990101'
GO
```

Next, a login is created based on the certificate. The login is granted both `ALTER TRACE` permission, as well as `AUTHENTICATE SERVER` permission, the latter of which gives it the right to propagate server-level permissions to database-level modules (such as the wrapper stored procedure):

```
CREATE LOGIN ALTER_TRACE_LOGIN
FROM CERTIFICATE ALTER_TRACE_CERT
GO
GRANT ALTER TRACE TO ALTER_TRACE_LOGIN
GO
GRANT AUTHENTICATE SERVER TO ALTER_TRACE_LOGIN
GO
```

Once that's taken care of, back up the certificate, including the private key. The backup will be used to restore the same certificate in any user database in which you'd like to use the wrapper stored procedure.

```
BACKUP CERTIFICATE ALTER_TRACE_CERT
TO FILE='C:\ALTER_TRACE.cer'
WITH PRIVATE KEY
(
    FILE='C:\ALTER_TRACE.pvk',
    ENCRYPTION BY PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~',
    DECRYPTION BY PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~'
)
GO
```

For the sake of this chapter's sample code, we'll show you how to build the wrapper procedure in `tempdb`, but in reality you could do this in any user database. The following code creates a certificate in `tempdb` from the backed-up version, then creates a simple wrapper over the `sp_trace_generateevent` stored procedure:

Code View:

```
USE tempdb
GO
CREATE CERTIFICATE ALTER_TRACE_CERT
FROM FILE='C:\ALTER_TRACE.cer'
WITH PRIVATE KEY
(
    FILE='C:\ALTER_TRACE.pvk',
    ENCRYPTION BY PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~',
    DECRYPTION BY PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~'
```

```

)
GO
CREATE PROCEDURE ThrowEvent
    @eventid INT,
    @userinfo nvarchar(128),
    @userdata varbinary(8000)
AS
BEGIN
    EXEC sp_trace_generateevent
        @eventid = @eventid,
        @userinfo = @userinfo,
        @userdata = @userdata
END
GO

```

To complete the exercise, the stored procedure is signed with the certificate which gives the procedure effectively all of the same permissions as the ALTER_TRACE_LOGIN login, and then permission is granted for any user to run the stored procedure:

```

ADD SIGNATURE TO ThrowEvent
BY CERTIFICATE ALTER_TRACE_CERT
WITH PASSWORD='-UsE_a!sTr0Ng_PwD-or-3~'
GO
GRANT EXEC ON ThrowEvent TO [public]
GO

```

Once the ThrowEvent stored procedure is created in the database(s) of your choice, you can begin using it from within other stored procedures, and, thanks to the certificate, you can do so without regard to what permissions the caller has. This can be an invaluable tool when trying to figure out the context around intermittent problems.

For example, suppose that you find during testing that one of your stored procedures that should be updating a few rows in a certain table seems to not update anything from time to time. This problem seems to be caused by the state of yet another table at exactly the time the stored procedure is called, but you haven't yet been able to reproduce things lining up in just that way.

To help debug this, you might insert the following code into your stored procedure, just after the update:

```

IF @@ROWCOUNT = 0
    EXEC ThrowEvent 82, N'No data inserted into MyTable', 0x0000

```

At this point, you would set up a trace to capture RPC:Starting and SQL:BatchStarting events, as well as the UserDefined:0 event. After letting it play for a while, once the user-defined event fires letting you know that no rows were inserted, you may have collected enough back data to figure out what state the other table was in at the exact time of the insert.

This is somewhat of a contrived example, but it helps show the power of adding this tool to your stored procedure debugging toolkit. The more controllable visibility you have into what's going on in your stored procedures, the easier it becomes to track down and fix small problems before they turn into major ones. In the next section, we'll get into how to keep SQL Trace itself from causing you problems.

Tracing Considerations and Design

Although SQL Trace was both designed and tested with high efficiency in mind, users must remember that it can be made to do quite a large amount of work. Collecting millions of events per minute on an extremely busy server can be taxing to all server resources— from memory to store the buffers, to network bandwidth if the rowset provider is used, to I/O if the file provider is used. This does not mean that SQL Trace should not be used on an active system; to the contrary, SQL Trace is a fantastic tool for tracking down issues even on the busiest of database servers. However, application of a bit of forethought and planning is necessary in order to ensure that your tracing session successfully answers your questions and does not end up causing you new problems.

The SQL Server Profiler Question

By far the easiest way to make sure that your trace activity does not cause issues on your busy production server is to avoid use of the SQL Server Profiler tool except as a scripting utility (to script trace definitions) and for very small trace jobs, such as when filtering for activity done by a single spid.

Advice against using SQL Server Profiler on a busy system is something we've heard for quite some time, but we were not sure just how much impact it would really have. SQL Server MVP and load testing expert Linchi Shea agreed to help us find an answer, and conducted a few tests using a TPC-C benchmark tool that he has written.

Linchi ran the tests on an HP DL585 G1 server with four single-core 2.6-GHz AMD Opteron processors. The server had 16 GB of RAM with 12 GB allocated for the SQL Server 2005 SP2 (9.00.3042) instance. The test database was 15 GB in total size, with 10 GB allocated for data and 5 GB allocated for logs. The actual data size was approximately 9 GB at the start of each batch of test runs and grew to approximately 12 GB over the course of the test. Workloads ranged from 20 to 300 users over the course of each test, with processors in the 75 to 85 percent range and I/O saturated at high user levels.

Three tests were run: one baseline test with no tracing enabled, one test using the default template with results returned to SQL Server Profiler, and a third test using the default template with a server-side trace (results output to a file). The results are fairly astounding, showing an almost 10 percent penalty in transactions per second throughput for the Profiler-based trace, with even a small number of emulated users, as shown in [Table 2-4](#).

Table 2-4. Results of Comparing Server-Side and Profiler-Based Traces

Number of Users	Average Transactions per Second	Average Response Time	Number of Emulated Users
Users	Profiler Trace	Server-Side Trace	Profiler Trace
No	Trace	Trace	Server-Side
20	41246374	1001015110631262001121120666593006501064194110	110

Linchi noted that the bandwidth utilization resulting from the Profiler trace was consuming over 35 percent of the 100 megabits available to his server, but he was unable to determine whether that was the actual cause of the slowdown. The good news is that Linchi found absolutely no performance difference between his baseline case and the test with the server-side trace. This test clearly indicates that server-side tracing is the correct approach for an already busy production environment, in order to avoid causing more issues.

Note



The numbers and data shown here tell only some of the story. For a complete write-up on these tests, please refer to Linchi's blog post on the topic, which can be found at the

following URL:

http://sqlblog.com/blogs/linchi_shea/archive/2007/08/01/trace-profiler-test.aspx.

It's important to remember that this test shows only a narrow view of a broad topic. It does not answer the question of when a trace will break a system that's already at the edge, or how much additional data can be collected before problems start. And it should be stressed that the results of this test should not be read to indicate that DBAs should go overboard and trace everything, server-side, without discretion. There will certainly be performance penalties, even with server-side traces, as the amount of data collected increases. However, the results do indicate that you should probably not worry too much if you do need to use SQL Trace to solve a problem, even on a production system.

Reducing Trace Overhead

Beyond simply not using SQL Server Profiler, there is quite a bit you can do to reduce the overhead of your traces. As shown in the preceding section, server-side traces are quite efficient, but that doesn't mean that you can go too crazy with them. The main thing you should try to avoid is overstressing the I/O system by collecting too much data, which will end up causing blocking conditions.

SQL Trace provides the ability to filter your traces—make sure to take advantage of it. Filters naturally reduce I/O by limiting the amount of data that is collected. When working on designing your traces, try to think about how you can filter them ahead of time—as part of the trace definition—instead of afterwards, once you load the data into a trace table.

Sometimes the best way to filter your trace to avoid excessive overhead is to actually create multiple traces, each with a different set of filters. For example, in the "Debugging Deadlocks" section, the trace shown was impossible to filter by spid because the Deadlock graph event may be fired on any number of system spids. Rather than creating a single monolithic trace, you might instead create one trace that includes the RPC:Starting and SQL:BatchStarting events, as well as Lock:Acquired and Lock:Released, and Lock:Escalation. This trace can be filtered based on the two spids you're focusing on.

A second, unfiltered trace can be simultaneously created, which includes only the Deadlock graph event. After capturing the required data, the events from both traces can be inserted into a single trace table, and treated as if they were captured by a single trace. If you also happened to capture the EventSequence column for all involved events, it will be even easier to piece things back together—and the combination of these two traces will not contain any results that you did not intend to capture.

By thinking somewhat laterally, it is possible to vastly reduce the I/O overhead of server-side tracing. Getting out of the mind-set of trying to squeeze everything into a single trace enables much more flexible use of filters and the ability to greatly narrow your trace's focus.

Max File Size, Rollover, and Data Collection

Another I/O-related consideration is the maximum file size parameter available when creating a server-side trace. When working with a new system that you suspect is especially busy, be very cautious about server-side traces because of the very real possibility that if you haven't been quite smart enough about your filters, the trace will quickly consume the drive on which you're collecting. Some systems are busy enough to capture 100 or more megabytes per second, just from the TextData columns of the RPC:Completed and SQL:BatchCompleted events.

In order to avoid this problem and test a trace, we generally start with a maximum file size of 50 megabytes, with rollover files turned off. We enable the trace and watch it, making sure that it doesn't immediately

consume the entire file and stop. If it does do so, it's time to re-think things and try again; if not, we keep going, perhaps setting a slightly larger maximum file size or configuring rollover files.

Rollover files can be incredibly useful if you'd like to run a longer-term server-side trace but still have some delayed visibility into data as it's collected. To set this up, configure your trace to use a small enough maximum file size that it will roll over on a regular basis—say, every 20 minutes, assuming that's the interval you'd like for your delayed data. Also configure the maximum number of rollover files to a large enough number to support the length of time you would like to run the trace.

Set up a SQL Server Agent job to periodically get the name of the current file, by querying the Path column of the sys.traces view. Check the folder that you're tracing into for older trace files—perhaps using xp_cmdshell—and insert these into the trace table. Don't forget to delete them when you're done.

Running the trace in this way will give you delayed visibility into the data, while keeping things fairly efficient and ensuring that you don't lose any events. Make sure to monitor the data collection from the trace tables and ensure that the insert itself isn't taking an excessively long time or tying up the I/O system too much. It can sometimes be difficult to balance the needs of production databases with the requirement of being able to monitor them.

Auditing: SQL Server's Built-in Traces

Aside from the user-defined traces we have discussed throughout this chapter, SQL Server uses tracing to help you audit user activity using a few built-in configurations. In this section, we will briefly discuss the default trace, blackbox traces, and Common Criteria traces—all part of the same SQL Trace infrastructure, but configured and used slightly differently than the traces discussed in the rest of the chapter.

Default Trace

Upon first being installed, SQL Server 2005 starts up a background trace known as the default trace. If left in place, this trace will have a trace ID of 1, and you can see it in the sys.traces view using the following query:

```
SELECT *
FROM sys.traces
WHERE id = 1
```

The trace is fairly lightweight, and includes events for monitoring server starts and stops, object creation and deletion, log and data file auto growth, and other database changes. The information collected by the trace can be retrieved using fn_trace_gettable, just like any other trace. In addition, data from the default trace is used for various reports that ship with SQL Server Management Studio. The trace is also useful as a general log in case something unexpected happens ("hey, who dropped my table!?"'), and for all of these reasons we recommend leaving it in place.

Should you feel the need to disable the default trace, you can do so by changing the default trace enabled server option, using the following Transact-SQL:

```
EXEC sp_configure 'default trace enabled', 0
RECONFIGURE
```

Note that this will immediately stop the trace; there is no need to restart the SQL Server service.

Blackbox Traces

Another background trace that SQL Server offers right out of the box is the so-called blackbox trace. This trace is designed to behave similarly to an airplane black box, to help you diagnose intermittent server crashes. It is quite a bit heavier than the default trace, and includes the SP:Starting, SQL:BatchStarting, Exception, and Attention events, with several columns from each event included in the trace.

This trace is configured by setting the @options parameter of sp_trace_create to a value of 8. The following Transact-SQL can be used to start a blackbox trace:

```
DECLARE @TraceId INT
EXEC sp_trace_create
    @TraceId OUTPUT,
    @options = 8
EXEC sp_trace_setstatus @TraceId, 1
```

The trace is automatically configured to use two rollover files, and flip back and forth between them when reaching its default maximum file size of 5 MB. However, some customers have reported that 5 MB was not enough back data to help them debug crash problems. To expand the size, you'll have to specify the @maxfilesize parameter, in addition to passing in a value for the @tracefile parameter (even a NULL). The following Transact-SQL code creates a blackbox trace with a 25 MB maximum size:

```
DECLARE @TraceId int
DECLARE @maxfilesize bigint
SET @maxfilesize = 25
EXEC sp_trace_create
    @TraceId OUTPUT,
    @options = 8,
    @tracefile = NULL,
    @maxfilesize = @maxfilesize
EXEC sp_trace_setstatus @TraceId, 1
```

By default, the path to the blackbox trace file is in the default SQL Server data file folder. However, just like the file size, the path can also be overridden if necessary, by using the @tracefile parameter.

In order to fully take advantage of the blackbox trace if you are facing intermittent problems, you want to make sure that it is always running when your server is running— including after either a planned or unplanned restart. To accomplish this, you can set the blackbox trace to start automatically when SQL Server starts. First, wrap the trace definition in a stored procedure in the master database:

```
USE master
GO
CREATE PROCEDURE StartBlackBoxTrace
AS
BEGIN
    DECLARE @TraceId int
    DECLARE @maxfilesize bigint
    SET @maxfilesize = 25
    EXEC sp_trace_create
        @TraceId OUTPUT,
        @options = 8,
        @tracefile = NULL,
        @maxfilesize = @maxfilesize
    EXEC sp_trace_setstatus @TraceId, 1
END
GO
```

Next, set the procedure to start automatically when the SQL Server service is started:

```
EXEC sp_procoption
'StartBlackBoxTrace', 'STARTUP', 'ON'
```

In the event of a crash, you can analyze the collected data to determine what stored procedures or queries were running at the point of failure and hopefully figure out what went wrong.

C2 and Common Criteria Auditing

SQL Server 2000 added a feature called C2 Audit, which allowed DBAs to meet U.S. government standards for auditing both unauthorized use of and damage to resources and data. A newer, international standard, called Common Criteria, is available in SQL Server 2005 (along with C2, which is still available).

Both of these audit modes utilize SQL Trace to capture a variety of data, including any logins, logouts, access to tables, and many other data points. Both also make use of the value of 4 passed to the @options parameter of sp_trace_create, which automatically shuts down the SQL Server service should any error occur writing to the trace file. This is intended to ensure that even in the case of disk error, none of the audit data will possibly be lost (with the server shut down, no audit data will be generated to lose).

Both C2 and Common Criteria audit modes are enabled or disabled using sp_configure. To enable C2 auditing, configure the C2 Audit Mode option to a value of 1. For Common Criteria, configure the Common Criteria Compliance Enabled option to a value of 1. In both cases, a service restart is required for the auditing to actually start.

Summary

SQL Trace is an extremely powerful tool for the SQL Server DBA and database developer, enabling a wide variety of monitoring, debugging, and auditing scenarios. In this chapter, we explored the internals of SQL Trace and the SQL Server Profiler user interface tool, as well as the security requirements for using them. Basic trace configuration using both Profiler and server-side traces was covered next. We then showed you how to use traces to solve various problems, and gave you some guidance on how to make sure that your tracing activities are as successful as possible. Finally, we discussed some of the traces that are shipped with SQL Server.

Properly harnessing what SQL Trace has to offer will make you a better, more effective DBA, and will enable you to quickly and effectively determine the root cause of many types of problems that would otherwise be extremely difficult to analyze. Think of SQL Trace as your constant companion as you work with SQL Server; knowing how to use it is one of those things that separates the good DBAs from the great ones.

Chapter 3. Query Execution

â Craig Freedman

In this chapter:	
Query Processing and Execution Overview	103

Reading Query Plans	107
Analyzing Plans	115
Summary	198

The SQL Server query processor consists of two components: the query optimizer and the query execution engine. The query optimizer is responsible for generating good query plans. The query execution engine takes the query plans generated by the query optimizer and, as its name suggests, runs them. Query execution involves many functions, including using the storage engine to retrieve and update data from tables and indexes and implementing operations such as joins and aggregation.

The focus of this chapter is on understanding query behavior by examining the details of your query execution plans. The chapter explains how the SQL Server query processor works, beginning with the basics of query plans and working toward progressively more complex examples.

Query Processing and Execution Overview

To better understand the factors that affect query performance, to understand how to spot potential performance problems with a query plan, and ultimately to learn how to use query optimizer hints to tune individual query plans, we first need to understand how the SQL Server query processor executes queries. In this section, we introduce iterators, one of the most fundamental query execution concepts, discuss how to read and understand query plans, explore some of the most common query execution operators, and learn how SQL Server combines these operators to execute even the most complex queries.

Iterators

SQL Server breaks queries down into a set of fundamental building blocks that we call operators or iterators. Each iterator implements a single basic operation such as scanning data from a table, updating data in a table, filtering or aggregating data, or joining two data sets. In all, there are a few dozen such primitive iterators. Iterators may have no children or may have one, two, or more children and can be combined into trees which we call query plans. By building appropriate query plans, SQL Server can execute any SQL statement. In practice, there are frequently many valid query plans for a given statement. The query optimizer's job is to find the best (for example, the cheapest or fastest) query plan for a given statement.

An iterator reads input rows either from a data source such as a table or from its children (if it has any) and produces output rows, which it returns to its parent. The output rows that an iterator produces depend on the operation that the iterator performs.

All iterators implement the same set of core methods. For example, the Open method tells an iterator to prepare to produce output rows, the GetRow method requests that an iterator produce a new output row, and the Close method indicates that the iterator's parent is through requesting rows. Because all iterators implement the same methods, iterators are independent of one another. That is, an iterator does not need specialized knowledge of its children (if any) or parent. Consequently, iterators can be easily combined in many different ways and into many different query plans.

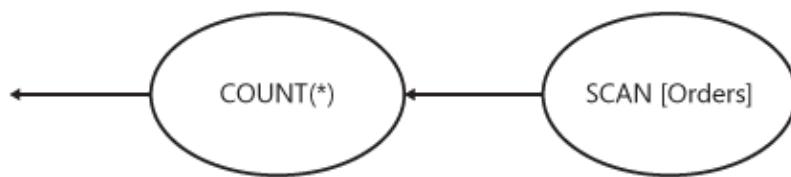
When SQL Server executes a query plan, control flows down the query tree. That is, SQL Server calls the methods Open and GetRow on the iterator at the root of the query tree and these methods propagate down through the tree to the leaf iterators. Data flows or more accurately is pulled up the tree when one iterator calls another iterator's GetRow method.

To understand how iterators work, let's look at an example. Most of the examples in this chapter, including the following example, are based on an extended version of the Northwind database, called Northwind2. You can download a script to build Northwind2 from the book's companion Web site. Consider this query:

```
SELECT COUNT(*) FROM [Orders]
```

The simplest way to execute this query is to scan each row in the Orders table and count the rows. SQL Server uses two iterators to achieve this result: one to scan the rows in the Orders table and another to count them, as illustrated in [Figure 3-1](#).

Figure 3-1. Iterators for basic COUNT(*) query



To execute this query plan, SQL Server calls Open on the root iterator in the plan which in this example is the COUNT(*) iterator. The COUNT(*) iterator performs the following tasks in the Open method:

1. Call Open on the scan iterator, which readies the scan to produce rows;
2. Call GetRow repeatedly on the scan iterator, counting the rows returned, and stopping only when GetRow indicates that it has returned all of the rows; and
3. Call Close on the scan iterator to indicate that it is done getting rows.

Note



COUNT(*) is actually implemented by the stream aggregate iterator, which we will describe in more detail later in this chapter.

Thus, by the time the COUNT(*) iterator returns from Open, it has already calculated the number of rows in the Orders table. To complete execution SQL Server calls GetRow on the COUNT(*) iterator and returns this result. [Technically, SQL Server calls GetRow on the COUNT(*) iterator one more time since it does not know that the COUNT(*) iterator produces only a single row until it tries to retrieve a second row. In response to the second GetRow call, the COUNT(*) iterator returns that it has reached the end of the result set.]

Note that the COUNT(*) iterator neither cares nor needs to know that it is counting rows from a scan iterator; it will count rows from any subtree that SQL Server puts below it, regardless of how simple or complex the subtree may be.

Properties of Iterators

Three important properties of iterators can affect query performance and are worth special attention. These properties are memory consumption, nonblocking vs. blocking, and dynamic cursor support.

Memory Consumption

All iterators require some small fixed amount of memory to store state, perform calculations, and so forth. SQL Server does not track this fixed memory or try to reserve this memory before executing a query. When SQL Server caches an executable plan, it caches this fixed memory so that it does not need to allocate it again and to speed up subsequent executions of the cached plan.

However, some iterators, referred to as memory-consuming iterators, require additional memory to execute. This additional memory is used to store row data. The amount of memory required by a memory-consuming operator is generally proportional to the number of rows processed. To ensure that the server does not run out of memory and that queries containing memory-consuming iterators do not fail, SQL Server estimates how much memory these queries need and reserves a memory grant before executing such a query.

Memory-consuming iterators can affect performance in a few ways.

1. Queries with memory-consuming iterators may have to wait to acquire the necessary memory grant and cannot begin execution if the server is executing other such queries and does not have enough available memory. This waiting can directly affect performance by delaying execution.
2. If too many queries are competing for limited memory resources, the server may suffer from reduced concurrency and/or throughput. This impact is generally not a major issue for data warehouses but is undesirable in OLTP (Online Transaction Processing) systems.
3. If a memory-consuming iterator requests too little memory, it may need to spill data to disk during execution. Spilling can have a significant adverse impact on the query and system performance because of the extra I/O overhead. Moreover, if an iterator spills too much data, it can run out of disk space on tempdb and fail.

The primary memory-consuming iterators are sort, hash join, and hash aggregation.

Nonblocking vs. Blocking Iterators

Iterators can be classified into two categories:

1. Iterators that consume input rows and produce output rows at the same time (in the GetRow method). We often refer to these iterators as "nonblocking."
2. Iterators that consume all input rows (generally in the Open method) before producing any output rows. We refer to these iterators as "blocking" or "stop-and-go."

The compute scalar iterator is a simple example of a nonblocking iterator. It reads an input row, computes a new output value using the input values from the current row, immediately outputs the new value, and continues to the next input row.

The sort iterator is a good example of a blocking iterator. The sort cannot determine the first output row until it has read and sorted all input rows. (The last input row could be the first output row; there is no way to know without first consuming every row.)

Blocking iterators often, but not always, consume memory. For example, as we just noted sort is both memory consuming and blocking. On the other hand, the COUNT(*) example, which we used to introduce the concept of iterators, does not consume memory and yet is blocking. It is not possible to know the number of rows

without reading and counting them all.

If an iterator has two children, the iterator may be blocking with respect to one and nonblocking with respect to the other. Hash join (which we'll discuss later in this chapter) is a good example of such an iterator.

Nonblocking iterators are generally optimal for OLTP queries where response time is important. They are often especially desirable for TOP N queries where N is small. Since the goal is to return the first few rows as quickly as possible, it helps to avoid blocking iterators, which might process more data than necessary before returning the first rows. Nonblocking iterators can also be useful when evaluating an EXISTS subquery, where it again helps to avoid processing more data than necessary to conclude that at least one output row exists.

Dynamic Cursor Support

The iterators used in a dynamic cursor query plan have special properties. Among other things, a dynamic cursor plan must be able to return a portion of the result set on each fetch request, must be able to scan forward or backward, and must be able to acquire scroll locks as it returns rows. To support this functionality, an iterator must be able to save and restore its state, must be able to scan forward or backward, must process one input row for each output row it produces, and must be nonblocking. Not all iterators have all of these properties.

For a query to be executed using a dynamic cursor, the optimizer must be able to find a query plan that uses only iterators that support dynamic cursors. It is not always possible to find such a plan. Consequently, some queries cannot be executed using a dynamic cursor. For example, queries that include a GROUP BY clause inherently violate the one input row for each output row requirement. Thus, such queries can never be executed using a dynamic cursor.

Chapter 3. Query Execution

â Craig Freedman

In this chapter:	
Query Processing and Execution Overview	103
Reading Query Plans	107
Analyzing Plans	115
Summary	198

The SQL Server query processor consists of two components: the query optimizer and the query execution engine. The query optimizer is responsible for generating good query plans. The query execution engine takes the query plans generated by the query optimizer and, as its name suggests, runs them. Query execution involves many functions, including using the storage engine to retrieve and update data from tables and indexes and implementing operations such as joins and aggregation.

The focus of this chapter is on understanding query behavior by examining the details of your query execution plans. The chapter explains how the SQL Server query processor works, beginning with the basics of query plans and working toward progressively more complex examples.

Query Processing and Execution Overview

To better understand the factors that affect query performance, to understand how to spot potential performance problems with a query plan, and ultimately to learn how to use query optimizer hints to tune individual query plans, we first need to understand how the SQL Server query processor executes queries. In this section, we introduce iterators, one of the most fundamental query execution concepts, discuss how to read and understand query plans, explore some of the most common query execution operators, and learn how SQL Server combines these operators to execute even the most complex queries.

Iterators

SQL Server breaks queries down into a set of fundamental building blocks that we call operators or iterators. Each iterator implements a single basic operation such as scanning data from a table, updating data in a table, filtering or aggregating data, or joining two data sets. In all, there are a few dozen such primitive iterators. Iterators may have no children or may have one, two, or more children and can be combined into trees which we call query plans. By building appropriate query plans, SQL Server can execute any SQL statement. In practice, there are frequently many valid query plans for a given statement. The query optimizer's job is to find the best (for example, the cheapest or fastest) query plan for a given statement.

An iterator reads input rows either from a data source such as a table or from its children (if it has any) and produces output rows, which it returns to its parent. The output rows that an iterator produces depend on the operation that the iterator performs.

All iterators implement the same set of core methods. For example, the Open method tells an iterator to prepare to produce output rows, the GetRow method requests that an iterator produce a new output row, and the Close method indicates that the iterator's parent is through requesting rows. Because all iterators implement the same methods, iterators are independent of one another. That is, an iterator does not need specialized knowledge of its children (if any) or parent. Consequently, iterators can be easily combined in many different ways and into many different query plans.

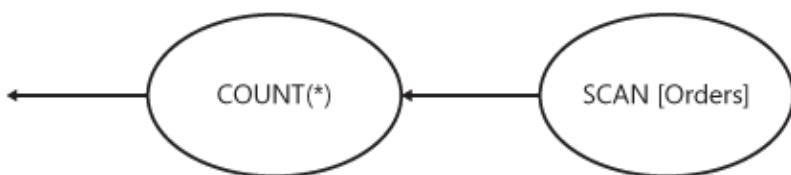
When SQL Server executes a query plan, control flows down the query tree. That is, SQL Server calls the methods Open and GetRow on the iterator at the root of the query tree and these methods propagate down through the tree to the leaf iterators. Data flows or more accurately is pulled up the tree when one iterator calls another iterator's GetRow method.

To understand how iterators work, let's look at an example. Most of the examples in this chapter, including the following example, are based on an extended version of the Northwind database, called Northwind2. You can download a script to build Northwind2 from the book's companion Web site. Consider this query:

```
SELECT COUNT(*) FROM [Orders]
```

The simplest way to execute this query is to scan each row in the Orders table and count the rows. SQL Server uses two iterators to achieve this result: one to scan the rows in the Orders table and another to count them, as illustrated in [Figure 3-1](#).

Figure 3-1. Iterators for basic COUNT(*) query



To execute this query plan, SQL Server calls Open on the root iterator in the plan which in this example is the COUNT(*) iterator. The COUNT(*) iterator performs the following tasks in the Open method:

1. Call Open on the scan iterator, which readies the scan to produce rows;
2. Call GetRow repeatedly on the scan iterator, counting the rows returned, and stopping only when GetRow indicates that it has returned all of the rows; and
3. Call Close on the scan iterator to indicate that it is done getting rows.

Note

 COUNT(*) is actually implemented by the stream aggregate iterator, which we will describe in more detail later in this chapter.

Thus, by the time the COUNT(*) iterator returns from Open, it has already calculated the number of rows in the Orders table. To complete execution SQL Server calls GetRow on the COUNT(*) iterator and returns this result. [Technically, SQL Server calls GetRow on the COUNT(*) iterator one more time since it does not know that the COUNT(*) iterator produces only a single row until it tries to retrieve a second row. In response to the second GetRow call, the COUNT(*) iterator returns that it has reached the end of the result set.]

Note that the COUNT(*) iterator neither cares nor needs to know that it is counting rows from a scan iterator; it will count rows from any subtree that SQL Server puts below it, regardless of how simple or complex the subtree may be.

Properties of Iterators

Three important properties of iterators can affect query performance and are worth special attention. These properties are memory consumption, nonblocking vs. blocking, and dynamic cursor support.

Memory Consumption

All iterators require some small fixed amount of memory to store state, perform calculations, and so forth. SQL Server does not track this fixed memory or try to reserve this memory before executing a query. When SQL Server caches an executable plan, it caches this fixed memory so that it does not need to allocate it again and to speed up subsequent executions of the cached plan.

However, some iterators, referred to as memory-consuming iterators, require additional memory to execute. This additional memory is used to store row data. The amount of memory required by a memory-consuming operator is generally proportional to the number of rows processed. To ensure that the server does not run out of memory and that queries containing memory-consuming iterators do not fail, SQL Server estimates how much memory these queries need and reserves a memory grant before executing such a query.

Memory-consuming iterators can affect performance in a few ways.

1. Queries with memory-consuming iterators may have to wait to acquire the necessary memory grant and cannot begin execution if the server is executing other such queries and does not have enough available memory. This waiting can directly affect performance by delaying execution.
2. If too many queries are competing for limited memory resources, the server may suffer from reduced concurrency and/or throughput. This impact is generally not a major issue for data warehouses but is undesirable in OLTP (Online Transaction Processing) systems.
3. If a memory-consuming iterator requests too little memory, it may need to spill data to disk during execution. Spilling can have a significant adverse impact on the query and system performance because of the extra I/O overhead. Moreover, if an iterator spills too much data, it can run out of disk space on tempdb and fail.

The primary memory-consuming iterators are sort, hash join, and hash aggregation.

Nonblocking vs. Blocking Iterators

Iterators can be classified into two categories:

1. Iterators that consume input rows and produce output rows at the same time (in the GetRow method). We often refer to these iterators as "nonblocking."
2. Iterators that consume all input rows (generally in the Open method) before producing any output rows. We refer to these iterators as "blocking" or "stop-and-go."

The compute scalar iterator is a simple example of a nonblocking iterator. It reads an input row, computes a new output value using the input values from the current row, immediately outputs the new value, and continues to the next input row.

The sort iterator is a good example of a blocking iterator. The sort cannot determine the first output row until it has read and sorted all input rows. (The last input row could be the first output row; there is no way to know without first consuming every row.)

Blocking iterators often, but not always, consume memory. For example, as we just noted sort is both memory consuming and blocking. On the other hand, the COUNT(*) example, which we used to introduce the concept of iterators, does not consume memory and yet is blocking. It is not possible to know the number of rows without reading and counting them all.

If an iterator has two children, the iterator may be blocking with respect to one and nonblocking with respect to the other. Hash join (which we'll discuss later in this chapter) is a good example of such an iterator.

Nonblocking iterators are generally optimal for OLTP queries where response time is important. They are often especially desirable for TOP N queries where N is small. Since the goal is to return the first few rows as quickly as possible, it helps to avoid blocking iterators, which might process more data than necessary before returning the first rows. Nonblocking iterators can also be useful when evaluating an EXISTS subquery, where it again helps to avoid processing more data than necessary to conclude that at least one output row exists.

Dynamic Cursor Support

The iterators used in a dynamic cursor query plan have special properties. Among other things, a dynamic cursor plan must be able to return a portion of the result set on each fetch request, must be able to scan forward or backward, and must be able to acquire scroll locks as it returns rows. To support this functionality, an iterator must be able to save and restore its state, must be able to scan forward or backward, must process

one input row for each output row it produces, and must be nonblocking. Not all iterators have all of these properties.

For a query to be executed using a dynamic cursor, the optimizer must be able to find a query plan that uses only iterators that support dynamic cursors. It is not always possible to find such a plan. Consequently, some queries cannot be executed using a dynamic cursor. For example, queries that include a GROUP BY clause inherently violate the one input row for each output row requirement. Thus, such queries can never be executed using a dynamic cursor.

Reading Query Plans

To better understand what the query processor is doing, we need a way to look at query plans. SQL Server 2005 has several different ways of displaying a query plan, and we refer to all these techniques collectively as "the showplan options."

Query Plan Options

SQL Server 2005 supports three showplan options: graphical, text, and XML. Graphical and text were available in prior versions of SQL Server; XML is new to SQL Server 2005. Each showplan option outputs the same query plan. The difference between these options is how the information is formatted, the level of detail included, how we read it, and how we can use it.

Graphical Plans

The graphical showplan option uses visually appealing icons that correspond to the iterators in the query plan. The tree structure of the query plan is clear. Arrows represent the data flow between the iterators. ToolTips provide detailed help, including a description of and statistical data on each iterator; this includes estimates of the number of rows generated by each operator (that is, the cardinality estimates), the average row size, and the cost of the operator. In SQL Server 2005, the Management Studio Properties window includes even more detailed information about each operator and about the overall query plan. Much of this data is new and was not available in SQL Server 2000. For example, the Properties window displays the SET options (such as ARITHABORT and ANSI_NULLS) used during the compilation of the plan, parameter and variable values used during optimization and at execution time, thread level execution statistics for parallel plans, the degree of parallelism for parallel plans, the size of the memory grant if any, the size of the cached query plan, requested and actual cursor types, information about query optimization hints, and information on missing indexes.

SQL Server 2005 SP2 adds compilation time (both elapsed and CPU time) and memory. Some of the available data varies from plan type to plan type and from operator to operator.

Generally, graphical plans give a good view of the big picture, which makes them especially useful for beginners and even for experienced users who simply want to browse plans quickly. On the other hand, some very large query plans are so large that they can only be viewed either by scaling the graphics down to a point where the icons are hard to read or by scrolling in two dimensions.

We can generate graphical plans using Management Studio in SQL Server 2005 (or using Query Analyzer in SQL Server 2000). Management Studio also supports saving and reloading graphical plans in files with a .sqlplan extension. In fact, the contents of a .sqlplan file are really just an XML plan and the same information is available in both graphical and XML plans. In prior versions of SQL Server, there is no way to save graphical plans (other than as an image file).

Text Plans

The text showplan option represents each iterator on a separate line. SQL Server uses indentation and vertical bars ("|" characters) to show the childâ– parent relationship between the iterators in the query tree. There are no explicit arrows, but data always flows up the plan from a child to a parent. Once you understand how to read it, text plans are often easier to readâ– especially when big plans are involved. Text plans can also be easier than graphical plans to save, manipulate, search, and/or compare, although many of these benefits are greatly diminished if not eliminated with the introduction of XML plans in SQL Server 2005.

There are two types of text plans. You can use SET SHOWPLAN_TEXT ON to display just the query plan. You can use SET SHOWPLAN_ALL ON to display the query plan along with most of the same estimates and statistics included in the graphical plan ToolTips and Properties windows.

XML Plans

The XML showplan option is new to SQL Server 2005. It brings together many of the best features of text and graphical plans. The ability to nest XML elements makes XML a much more natural choice than text for representing the tree structure of a query plan. XML plans comply with a published XSD schema (<http://schemas.microsoft.com/sqlserver/2004/07/showplan/showplanxml.xsd>) and, unlike text and graphical plans, are easy to search and process programmatically using any standard XML tools. You can even save XML plans in a SQL Server 2005 XML column, index them, and query them using SQL Server 2005's built-in XQuery functionality. Moreover, while compared with text plans the native XML format is more challenging to read directly, as noted previously, Management Studio can save graphical showplan output as XML plan files (with the .sqlplan extension) and can load XML plan files (again with the .sqlplan extension) and display them graphically.

XML plans contain all of the information available in SQL Server 2000 via either graphical or text plans. In addition, XML plans include the same detailed new information mentioned previously that is available using graphical plans and the Management Studio Properties window. XML plans are also the basis for the new USEPLAN query hint described in [Chapters 4](#) and [5](#).

The XML plan follows a hierarchy of a batch element, a statement element, and a query plan element (<QueryPlan>). If a batch or procedure contains multiple statements, the XML plan output for that batch or procedure will contain multiple query plans. Within the query plan element is a series of relational operator elements (<RelOp>). There is one relational operator element for each iterator in the query plan, and these elements are nested according to the tree structure of the query plan. Like the other showplan options, each relational operator element includes cost estimates and statistics, as well as some operator-specific information.

Estimated vs. Actual Query Plans

We can ask SQL Server to output a plan (for any showplan optionâ– graphical, text, or XML) with or without actually running a query.

We refer to a query plan generated without executing a query as the "estimated execution plan" as SQL Server may choose to recompile the query (recompiles may occur for a variety of reasons) and may generate a different query plan at execution time. The estimated execution plan is useful for a variety of purposes, such as viewing the query plan of a long-running query without waiting for it to complete; viewing the query plan for an insert, update, or delete statement without altering the state of the database or acquiring any locks; or exploring the effect of various optimization hints on a query plan without actually running the query. The estimated execution plan includes cardinality, row size, and cost estimates.

Tip

The estimated costs reported by the optimizer are intended as a guide to compare the anticipated relative cost of various operators within a single query plan or the relative cost of two different plans. These estimates are unitless and are not meant to be interpreted in any absolute sense such as milliseconds or seconds.

We refer to a query plan generated after executing a query as the "actual execution plan." The actual execution plan includes the same information as the estimated execution plan plus the actual row counts and the actual number of executions for each operator. By comparing the estimated and actual row counts, we can identify cardinality estimation errors, which may lead to other plan issues. XML plans include even more information, such as actual parameter and variable values at execution time; the memory grant and degree of parallelism if appropriate; and thread level row, execution, rewind, and rebind counts. (We cover rewinds and rebinds later in this chapter.)

Tip

The actual execution plan includes the same cost estimates as the estimated execution plan. Although SQL Server actually executes the query plan while generating the actual execution plan, these cost estimates are still the same estimates generated by the optimizer and do not reflect the actual execution cost.

There are several Transact-SQL commands that we can use to collect showplan option output when running ad hoc queries from SQL Server Management Studio or from the SQLCMD command line utility. These commands allow us to collect both text and XML plans, as well as estimated and actual plans. [Table 3-1](#) lists all of the available SET commands to enable showplan options.

Table 3-1. SET Commands for Displaying Query Plans

Command	Execute Query?	Include Estimated Row Counts & Stats	Include Actual Row Counts & Stats
SET SHOWPLAN_TEXT	ON	No	No
SET SHOWPLAN_ALL	ON	No	Yes
SET STATISTICS PROFILE	ON	Yes	Yes
SET SHOWPLAN_XML	ON	No	Yes
SET STATISTICS PROFILE XML	ON	Yes	Yes

We can also collect all forms of query plans using SQL Trace and XML plans using Dynamic Management Views (DMVs) (which are new to SQL Server 2005). These options are especially useful when analyzing applications in which you do not have access to the source code. Obtaining plan information from traces is discussed in [Chapter 2](#), "Tracing and Profiling." The DMVs that contain plan information are discussed in [Chapter 5](#), "Plan Caching and Recompilation."

Query Plan Display Options

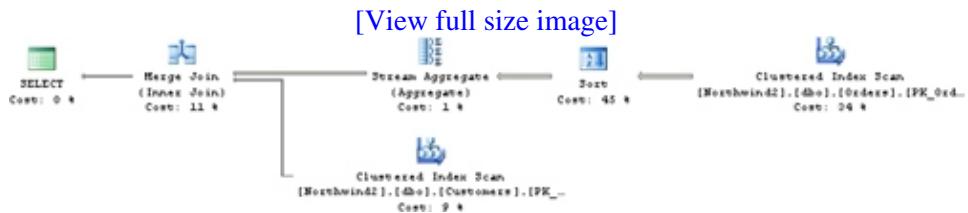
Let's compare the various ways of viewing query plans. As an example, consider the following query:

```
DECLARE @Country nvarchar(15)
SET @Country = N'USA'
SELECT O.[CustomerId], MAX(O.[Freight]) AS MaxFreight
FROM [Customers] C JOIN [Orders] O
  ON C.[CustomerId] = O.[CustomerId]
WHERE C.[Country] = @Country
GROUP BY O.[CustomerId]
```

```
OPTION (OPTIMIZE FOR (@Country = N'UK'))
```

The graphical plan for this query is shown in [Figure 3-2](#).

[Figure 3-2. A graphical execution plan](#)



Do not be too concerned at this point with understanding how the operators in this query plan actually function. Later in this chapter, we will delve into the details of the various operators. For now, simply observe how SQL Server combines the individual operators together in a tree structure. Notice that the clustered index scans are leaf operators and have no children, the sort and stream aggregate operators have one child each, and the merge join operator has two children. Also, notice how the data flows as shown by the arrows from the leaf operators on the right side of the plan to the root of the tree on the left side of the plan.

[Figure 3-3](#) shows the ToolTip information and [Figure 3-4](#) shows the Properties window from the actual (runtime) plan for the merge join operator. The ToolTip and Properties window show additional information about the operator, the optimizer's cost and cardinality estimates, and the actual number of output rows.

[Figure 3-3. ToolTip for merge join operator in a graphical plan](#)

Merge Join	
Match rows from two suitably sorted input tables exploiting their sort order.	
Physical Operation	Merge Join
Logical Operation	Inner Join
Actual Number of Rows	13
Estimated I/O Cost	0
Estimated CPU Cost	0.0058023
Estimated Operator Cost	0.005849 (11%)
Estimated Subtree Cost	0.0534411
Estimated Number of Rows	7
Estimated Row Size	25 B
Actual Rebinds	0
Actual Rewinds	0
Many to Many	False
Node ID	1
Where (join columns)	
([Northwind2].[dbo].[Customers].CustomerID) = ([Northwind2].[dbo].[Orders].CustomerID)	
Output List	
[Northwind2].[dbo].[Orders].CustomerID, Expr1004	

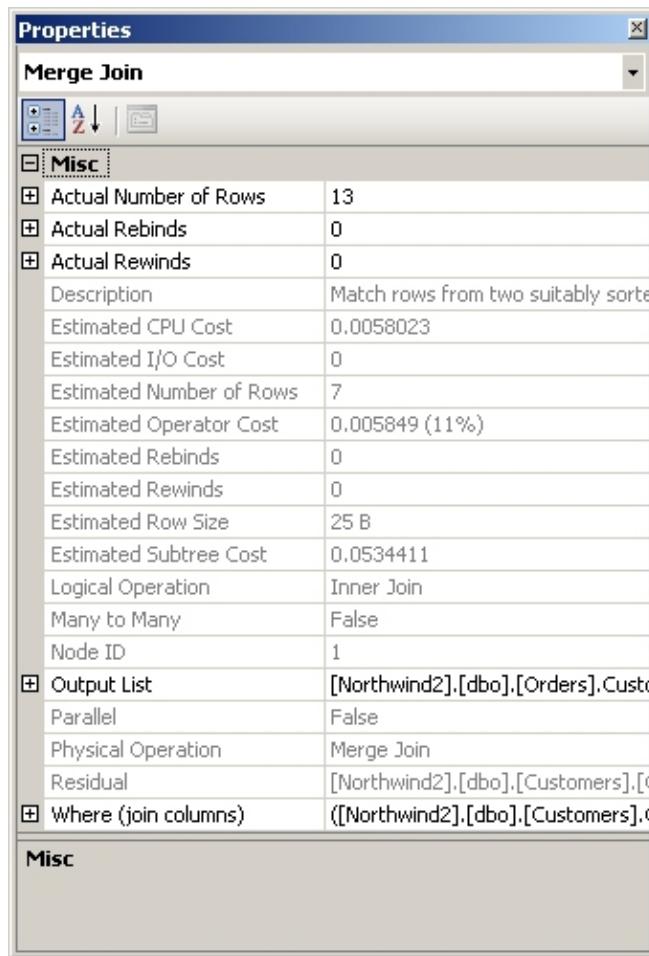
Figure 3-4. Properties window for merge join operator

Figure 3-5 shows the Properties window for the SELECT icon at the root of the plan. Note that it includes query-wide information such as the SET options used during compilation, the compilation time and memory, the cached plan size, the degree of parallelism, the memory grant, and the parameter and variable values used during compilation and execution. We will discuss the meaning of these fields as part of the XML plan example below. Keep in mind that a variable and a parameter are very different elements and the difference will be discussed in detail in [Chapter 5](#). However, the various query plans that we will examine use the term parameter to refer to either variables or parameters.

Figure 3-5. Properties window for SELECT at the top of a query plan

Properties	
SELECT	
	Z
Misc	
Cached plan size	15 B
CompileCPU	42
CompileMemory	280
CompileTime	209
Degree of Parallelism	0
Estimated Number of Rows	46,4367
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0534411
Logical Operation	
Memory Grant	64
Optimization Level	FULL
Parameter List	@Country
Column	@Country
Parameter Compiled Value	N'UK'
Parameter Runtime Value	N'USA'
Physical Operation	
Reason For Early Termination	Good Enough Plan Found
Set Options	CONCAT_NULL_YIELDS_NULL: False
ANSI_NULLS	False
ANSI_PADDING	False
ANSI_WARNINGS	False
ARITHABORT	True
CONCAT_NULL_YIELDS_NU	False
NUMERIC_ROUNDABORT	False
QUOTED_IDENTIFIER	False
Statement	SELECT O.[CustomerId], MAX(O.[Freight]) AS [Max Freight] FROM [Customers] C JOIN [Orders] O ON C.CustomerID = O.CustomerID WHERE C.Country = @Country
Misc	

Now let's consider the same query plan by looking at the output of SET SHOWPLAN_TEXT ON. Here is the text plan showing the query plan only:

Code View:

```
--Merge Join(Inner Join, MERGE:([O].[CustomerID])=([C].[CustomerID]), RESIDUAL:(...))
--Stream Aggregate(GROUP BY:([O].[CustomerID])
    DEFINE:([Expr1004]=MAX([O].[Freight])))
|   |--Sort(ORDER BY:([O].[CustomerID] ASC))
|       |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]),
    WHERE:([C].[Country]=[@Country]) ORDERED FORWARD)
```

Note



This plan and all of the other text plan examples in this chapter and in [Chapter 4](#) "Troubleshooting Query Performance," have been edited for brevity and to improve clarity. For instance, the database and schema name of objects have been removed from all plans. In some cases, lines have been wrapped, where they wouldn't normally wrap in the output.

Notice how, while there are no icons or arrows, this view of the plan has precisely the same operators and tree structure as the graphical plan. Recall that each line represents one operator—the equivalent of one icon in the graphical plan—and the vertical bars (the "I" characters) link each operator to its parent and children.

The output of SET SHOWPLAN_ALL ON includes the same plan text but, as noted previously, also includes additional information including cardinality and cost estimates. The SET STATISTICS PROFILE ON output includes actual row and operator execution counts, in addition to all of the other information.

Finally, here is a highly abbreviated version of the SET STATISTICS XML ON output for the same query plan. Notice how we have the same set of operators in the XML version of the plan as we did in the graphical and text versions. Also observe how the child operators are nested within the parent operator's XML element. For example, the merge join has two children and, thus, there are two relational operator elements nested within the merge join's relational operator element.

Code View:

```
<StmtSimple StatementText=
    "SELECT O.[CustomerId], MAX(O.[Freight]) as MaxFreight
     FROM [Customers] C JOIN [Orders] O
       ON C.[CustomerId] = O.[CustomerId]
      WHERE C.[Country] = @Country
     GROUP BY O.[CustomerId]
    OPTION (OPTIMIZE FOR (@Country = N'UK'))"...
<StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"
                     CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false"
                     ANSI_PADDING="false" ANSI_WARNINGS="false"
                     NUMERIC_ROUNDABORT="false" />
<QueryPlan DegreeOfParallelism="0" MemoryGrant="64" CachedPlanSize="15"
            CompileTime="20" CompileCPU="20" CompileMemory="280">
  <RelOp NodeId="1" PhysicalOp="Merge Join" LogicalOp="Inner Join"...
    <Merge ManyToMany="0">
      <RelOp NodeId="2" PhysicalOp="Stream Aggregate" LogicalOp="Aggregate"...
        <StreamAggregate>
          <RelOp NodeId="3" PhysicalOp="Sort" LogicalOp="Sort"...
            <MemoryFractions Input="1" Output="1" />
            <Sort Distinct="0">
              <RelOp NodeId="4" PhysicalOp="Clustered Index Scan"
                    LogicalOp="Clustered Index Scan"...
                <IndexScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
                  <Object Database="[Northwind2]" Schema="[dbo]" Table="[Orders]"
                    Index="[PK_Orders]" Alias="[O]" />
                </IndexScan>
              </RelOp>
            </Sort>
          </RelOp>
        </StreamAggregate>
      </RelOp>
      <RelOp NodeId="8" PhysicalOp="Clustered Index Scan"
            LogicalOp="Clustered Index Scan"...
        <IndexScan Ordered="1" ScanDirection="FORWARD" ForcedIndex="0"
          NoExpandHint="0">
          <Object Database="[Northwind2]" Schema="[dbo]" Table="[Customers]"
            Index="[PK_Customers]" Alias="[C]" />
        <Predicate>
          <ScalarOperator ScalarString="[Northwind2].[dbo].[Customers].[Country]
                                         as [C].[Country]=[@Country]">
```

```

        </ScalarOperator>
    </Predicate>
</IndexScan>
</RelOp>
</Merge>
</RelOp>
<ParameterList>
    <ColumnReference Column="@Country" ParameterCompiledValue="N'UK'" 
        ParameterRuntimeValue="N'USA" />
</ParameterList>
</QueryPlan>
</StmtSimple>

```

There are some other elements worth pointing out:

- The `<StmtSimple>` element includes a `StatementText` attribute, which as one might expect, includes the original statement text. Depending on the statement type, the `<StmtSimple>` element may be replaced by another element such as `<StmtCursor>`.
- The `<StatementSetOptions>` element includes attributes for the various SET options.
- The `<QueryPlan>` element includes the following attributes:
 - ◆ `DegreeOfParallelism`: The number of threads per operator for a parallel plan. A value of zero or one indicates a serial plan. This example is a serial plan.
 - ◆ `MemoryGrant`: The total memory granted to run this query in 2-Kbyte units. (The memory grant unit is documented as Kbytes in the showplan schema but is actually reported in 2-Kbyte units.) This query was granted 128 Kbytes.
 - ◆ `CachedPlanSize`: The amount of plan cache memory (in Kbytes) consumed by this query plan.
 - ◆ `CompileTime` and `CompileCPU`: The elapsed and CPU time (in milliseconds) used to compile this plan. (These attributes are new in SQL Server 2005 SP2.)
 - ◆ `CompileMemory`: The amount of memory (in Kbytes) used while compiling this query. (This attribute is new in SQL Server 2005 SP2.)
- The `<QueryPlan>` element also includes a `<ParameterList>` element, which includes the compile time and run-time values for each parameter and variable. In this example, there is just the one `@Country` variable.
- The `<RelOp>` element for each memory-consuming operator (in this example just the sort) includes a `<MemoryFractions>` element, which indicates the portion of the total memory grant used by that operator. There are two fractions. The input fraction refers to the portion of the memory grant used while the operator is reading input rows. The output fraction refers to the portion of the memory grant used while the operator is producing output rows. Generally, during the input phase of an operator's execution, it must share memory with its children; during the output phase of an operator's execution, it must share memory with its parent. Since in this example, the sort is the only memory-consuming operator in the plan, it uses the entire memory grant. Thus, the fractions are both one.

Although they have been truncated from the above output, each of the relational operator elements includes additional attributes and elements with all of the estimated and run-time statistics available in the graphical and text query plan examples:

```

<RelOp NodeId="1" PhysicalOp="Merge Join" LogicalOp="Inner Join"
    EstimateRows="7" EstimateIO="0"
    EstimateCPU="0.0058023" AvgRowSize="25"
    EstimatedTotalSubtreeCost="0.0534411" Parallel="0"
    EstimateRebinds="0" EstimateRewinds="0">
<RunTimeInformation>

```

```

<RunTimeCountersPerThread Thread="0" ActualRows="13"
    ActualEndOfScans="1" ActualExecutions="1" />
</RunTimeInformation>
...
</RelOp>
```

Note



Most of the examples in this chapter display the query plan in text format, obtained with SET SHOWPLAN_TEXT ON. Text format is more compact and easier to read than XML format and also includes more detail than screenshots of plans in graphical format. However, in some cases it is important to observe the "shape" of a query plan, and we will be showing you some examples of graphical plans. If you prefer to see plans in a format other than the one supplied in this chapter, you can download the code for the queries in this chapter from the companion Web site, and display the plans in the format of your choosing using your own SQL Server Management Studio.

Analyzing Plans

To really understand query plans and to really be able to spot, fix, or work around problems with query plans, we need a solid understanding of the query operators that make up these plans. All in all, there are too many operators to discuss them in one chapter. Moreover, there are innumerable ways to combine these operators into query plans. Thus, in this section, we focus on understanding the most common query operators—the most basic building blocks of query execution—and give some insight into when and how SQL Server uses them to construct a variety of interesting query plans. Specifically, we will look at scans and seeks, joins, aggregations, unions, a selection of subquery plans, and parallelism. With an understanding of how these basic operators and plans work, it is possible to break down and understand much bigger and more complex query plans.

Scans and Seeks

Scans and seeks are the iterators that SQL Server uses to read data from tables and indexes. These iterators are among the most fundamental ones that SQL Server supports. They appear in nearly every query plan. It is important to understand the difference between scans and seeks: a scan processes an entire table or the entire leaf level of an index, whereas a seek efficiently returns rows from one or more ranges of an index based on a predicate.

Let's begin by looking at an example of a scan. Consider the following query:

```
SELECT [OrderId] FROM [Orders] WHERE [RequiredDate] = '1998-03-26'
```

We have no index on the RequiredDate column. As a result, SQL Server must read every row of the Orders table, evaluate the predicate on RequiredDate for each row, and, if the predicate is true (that is, if the row qualifies), return the row.

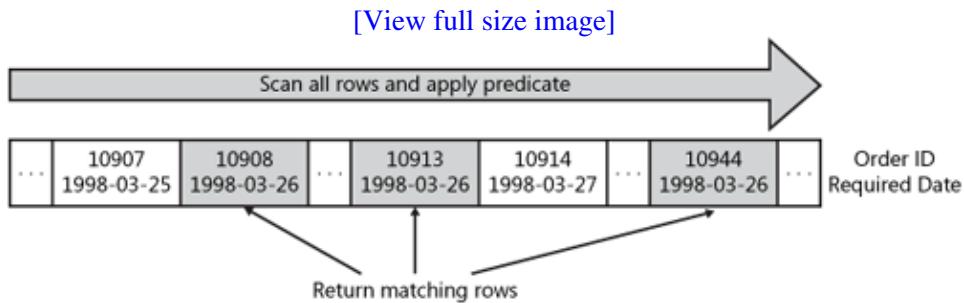
To maximize performance, whenever possible, SQL Server evaluates the predicate in the scan iterator. However, if the predicate is too complex or too expensive, SQL Server may evaluate it in a separate filter iterator. The predicate appears in the text plan with the WHERE keyword or in the XML plan with the

<Predicate> tag. Here is the text plan for the above query:

```
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]),
 WHERE:([Orders].[RequiredDate]='1998-03-26'))
```

[Figure 3-6](#) illustrates a scan:

Figure 3-6. A scan operation examines all the rows in all the pages of a table



Since a scan touches every row in the table whether or not it qualifies, the cost is proportional to the total number of rows in the table. Thus, a scan is an efficient strategy if the table is small or if many of the rows qualify for the predicate. However, if the table is large and if most of the rows do not qualify, a scan touches many more pages and rows and performs many more I/Os than is necessary.

Now let's look at an example of an index seek. Suppose we have a similar query, but this time the predicate is on the OrderDate column on which we do have an index:

```
SELECT [OrderId] FROM [Orders] WHERE [OrderDate] = '1998-02-26'
```

This time SQL Server is able to use the index to navigate directly to those rows that satisfy the predicate. In this case, we refer to the predicate as a seek predicate. In most cases, SQL Server does not need to evaluate the seek predicate explicitly; the index ensures that the seek operation only returns rows that qualify. The seek predicate appears in the text plan with the SEEK keyword or in the XML plan with the <SeekPredicates> tag. Here is the text plan for this example:

Code View:

```
--Index Seek(OBJECT:([Orders].[OrderDate]),
 SEEK:([Orders].[OrderDate]=CONVERT_IMPLICIT(datetime,[@1],0)) ORDERED FORWARD)
```

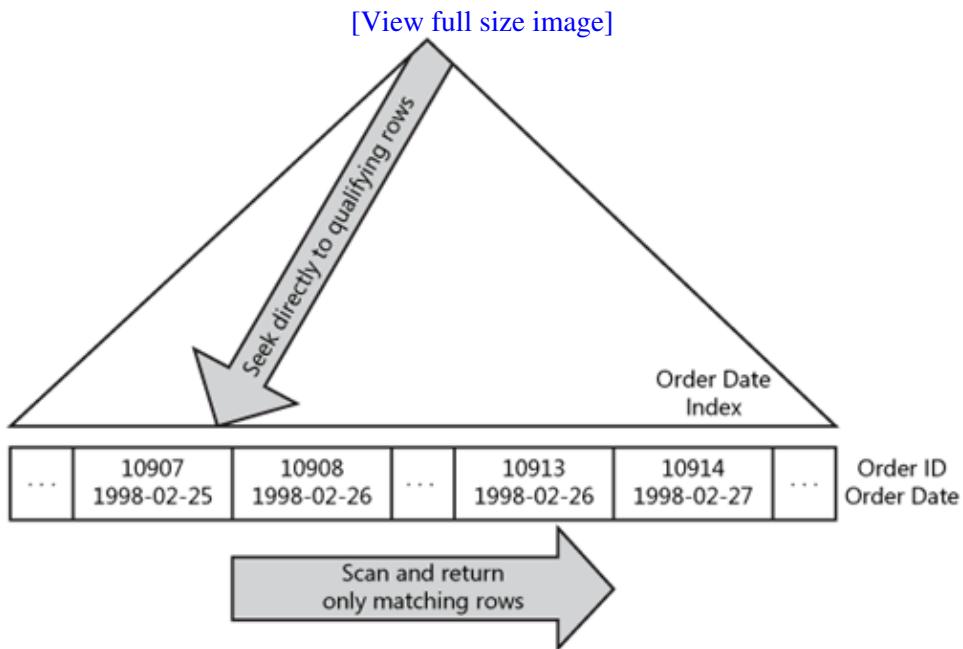
Note



Notice that SQL Server autoparameterized the query by substituting the parameter @1 for the literal date.

Figure 3-7 illustrates an index seek:

Figure 3-7. An index seek starts at the root and navigates to the leaf to find qualifying rows



Since a seek only touches rows that qualify and pages that contain these qualifying rows, the cost is proportional to the number of qualifying rows and pages rather than to the total number of rows in the table. Thus, a seek is generally a more efficient strategy if we have a highly selective seek predicate; that is, if we have a seek predicate that eliminates a large fraction of the table.

SQL Server distinguishes between scans and seeks as well as between scans on heaps (an object with no clustered index), scans on clustered indexes, and scans on nonclustered indexes. [Table 3-2](#) shows how all of the valid combinations appear in plan output.

Table 3-2. Scan and Seek Operators as They Appear in a Query Plan

Scan	Seek	Heap	Table Scan	Clustered Index Scan	Clustered Index Seek	Nonclustered Index Scan	Index Seek
-------------	-------------	-------------	-------------------	-----------------------------	-----------------------------	--------------------------------	-------------------

Seekable Predicates and Covered Columns

Before SQL Server can perform an index seek, it must determine whether the keys of the index are suitable for evaluating a predicate in the query. We refer to a predicate that may be used as the basis for an index seek as a "seekable predicate." SQL Server must also determine whether the index contains or "covers" the set of the columns that are referenced by the query. The following discussion explains how to determine which predicates are seekable, which predicates are not seekable, and which columns an index covers.

Single-Column Indexes

Determining whether a predicate can be used to seek on a single-column index is fairly straightforward. SQL Server can use single-column indexes to answer most simple comparisons including equality and inequality (greater than, less than, etc.) comparisons. More complex expressions, such as functions over a column and LIKE predicates with a leading wildcard character, will generally prevent SQL Server from using an index seek.

For example, suppose we have a single-column index on a column Col1. We can use this index to seek on these predicates:

- [Col1] = 3.14
- [Col1] > 100
- [Col1] BETWEEN 0 AND 99
- [Col1] LIKE 'abc%'
- [Col1] IN (2, 3, 5, 7)

However, we cannot use the index to seek on these predicates:

- ABS([Col1]) = 1
- [Col1] + 1 = 9
- [Col1] LIKE '%abc'

Composite Indexes

Composite, or multicolumn, indexes are slightly more complex. With a composite index, the order of the keys matters. It determines the sort order of the index, and it affects the set of seek predicates that SQL Server can evaluate using the index.

For an easy way to visualize why order matters, think about a phone book. A phone book is like an index with the keys (last name, first name). The contents of the phone book are sorted by last name, and we can easily look someone up if we know their last name. However, if we have only a first name, it is very difficult to get a list of people with that name. We would need another phone book sorted on first name.

In the same way, if we have an index on two columns, we can only use the index to satisfy a predicate on the second column if we have an equality predicate on the first column. Even if we cannot use the index to satisfy the predicate on the second column, we may be able to use it on the first column. In this case, we introduce a "residual" predicate for the predicate on the second column. This predicate is evaluated just like any other scan predicate.

For example, suppose we have a two-column index on columns Col1 and Col2. We can use this index to seek on any of the predicates that worked on the single-column index. We can also use it to seek on these additional predicates:

- [Col1] = 3.14 AND [Col2] = 'pi'
- [Col1] = 'xyzzy' AND [Col2] <= 0

For the next set of examples, we can use the index to satisfy the predicate on column Col1, but not on column Col2. In these cases, we need a residual predicate for column Col2.

- [Col1] > 100 AND [Col2] > 100
- [Col1] LIKE 'abc%' AND [Col2] = 2

Finally, we cannot use the index to seek on the next set of predicates as we cannot seek even on column Col1. In these cases, we must use a different index (that is, one where column Col2 is the leading column) or we must use a scan with a predicate.

- [Col2] = 0
- [Col1] + 1 = 9 AND [Col2] BETWEEN 1 AND 9
- [Col1] LIKE '%abc' AND [Col2] IN (1, 3, 5)

Identifying an Index's Keys

In most cases, the index keys are the set of columns that you specify in the CREATE INDEX statement. However, when you create a nonunique nonclustered index on a table with a clustered index, we append the clustered index keys to the nonclustered index keys if they are not explicitly part of the nonclustered index keys. You can seek on these implicit keys as if you specified them explicitly.

Covered Columns

The heap or clustered index for a table (often called the "base table") contains (or "covers") all columns in the table. Nonclustered indexes, on the other hand, contain (or cover) only a subset of the columns in the table. By limiting the set of columns stored in a nonclustered index, SQL Server can store more rows on each page, which saves disk space and improves the efficiency of seeks and scans by reducing the number of I/Os and the number of pages touched. However, a scan or seek of an index can only return the columns that the index covers.

Each nonclustered index covers the key columns that were specified when it was created. Also, if the base table is a clustered index, each nonclustered index on this table covers the clustered index keys regardless of whether they are part of the nonclustered index's key columns. In SQL Server 2005, we can also add additional nonkey columns to a nonclustered index using the INCLUDE clause of the CREATE INDEX statement. Note that unlike index keys, order is not relevant for included columns.

Example of Index Keys and Covered Columns

For example, given this schema:

```
CREATE TABLE T_heap (a int, b int, c int, d int, e int, f int)
CREATE INDEX T_heap_a ON T_heap (a)
CREATE INDEX T_heap_bc ON T_heap (b, c)
CREATE INDEX T_heap_d ON T_heap (d) INCLUDE (e)
CREATE UNIQUE INDEX T_heap_f ON T_heap (f)

CREATE TABLE T_clu (a int, b int, c int, d int, e int, f int)
CREATE UNIQUE CLUSTERED INDEX T_clu_a ON T_clu (a)
CREATE INDEX T_clu_b ON T_clu (b)
CREATE INDEX T_clu_ac ON T_clu (a, c)
CREATE INDEX T_clu_d ON T_clu (d) INCLUDE (e)
CREATE UNIQUE INDEX T_clu_f ON T_clu (f)
```

The key columns and covered columns for each index are shown in [Table 3-3](#).

Table 3-3. Key Columns and Covered Columns in a Set of Nonclustered Indexes

IndexKey	ColumnsCovered	Columns
T_clu_ac	T_heap_aaa	T_heap_bcb, cb, cT_heap_ddd, eT_heap_fff
T_clu_dd	T_clu_aaa	b, bT_clu_aca, ca, cT_clu_dd, aa, d, eT_clu_ffa, f
T_clu_f		

Note that the key columns for each of the nonclustered indexes on T_clu include the clustered index key column a with the exception of T_clu_f, which is a unique index. T_clu_ac includes column a explicitly as the first key column of the index, and so the column appears in the index only once and is used as the first key column. The other indexes do not explicitly include column a, so the column is merely appended to the end of the list of keys.

Bookmark Lookup

We've just seen how SQL Server can use an index seek to efficiently retrieve data that matches a predicate on the index keys. However, we also know that nonclustered indexes do not cover all of the columns in a table. Suppose we have a query with a predicate on a nonclustered index key that selects columns that are not covered by the index. If SQL Server performs a seek on the nonclustered index, it will be missing some of the required columns. Alternatively, if it performs a scan of the clustered index (or heap), it will get all of the columns, but will touch every row of the table and the operation will be less efficient. For example, consider the following query:

Code View:

```
SELECT [OrderId], [CustomerId] FROM [Orders] WHERE [OrderDate] = '1998-02-26'
```

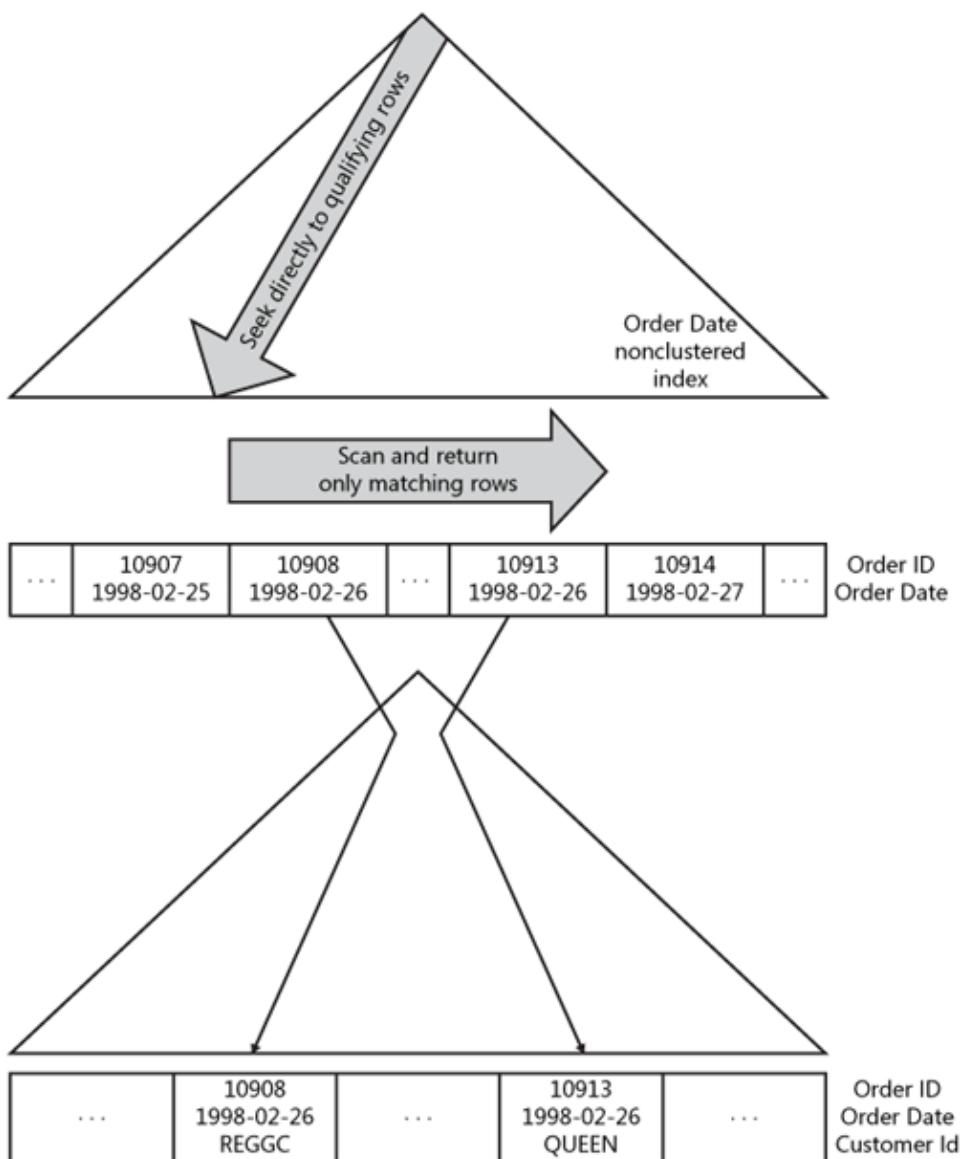
This query is identical to the query we used earlier to illustrate an index seek, but this time the query selects two columns: OrderId and CustomerId. The nonclustered index OrderDate only covers the OrderId column (which also happens to be the clustering key for the Orders table in the Northwind2 database).

SQL Server has a solution to this problem. For each row that it fetches from the nonclustered index, it can look up the value of the remaining columns (for instance, the CustomerId column in our example) in the clustered index. We call this operation a "bookmark lookup." A bookmark is a pointer to the row in the heap or clustered index. SQL Server stores the bookmark for each row in the nonclustered index precisely so that it can always navigate from the nonclustered index to the corresponding row in the base table.

[Figure 3-8](#) illustrates a bookmark lookup from a nonclustered index to a clustered index.

Figure 3-8. A bookmark lookup uses the information from the nonclustered index leaf level to find the row in the clustered index

[\[View full size image\]](#)

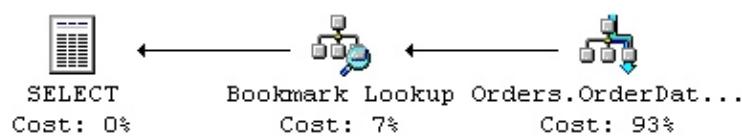


SQL Server 2000 implemented bookmark lookup using a dedicated iterator. The text plan shows us the index seek and bookmark lookup operators, as well as indicating the column used for the seek:

```
|--Bookmark Lookup(BMKMARK:([Bmk1000]), OBJECT:([Orders]))
|--Index Seek(OBJECT:([Orders].[OrderDate]),
SEEK:([Orders].[OrderDate]=Convert([@1])) ORDERED FORWARD)
```

The graphical plan is shown in [Figure 3-9](#).

Figure 3-9. Graphical plan for index seek and bookmark lookup in SQL Server 2000



The SQL Server 2005 plan for the same query uses a nested loops join (we will explain the behavior of this operator later in this chapter) combined with a clustered index seek, if the base table is a clustered index, or a RID (row id) lookup if the base table is a heap.

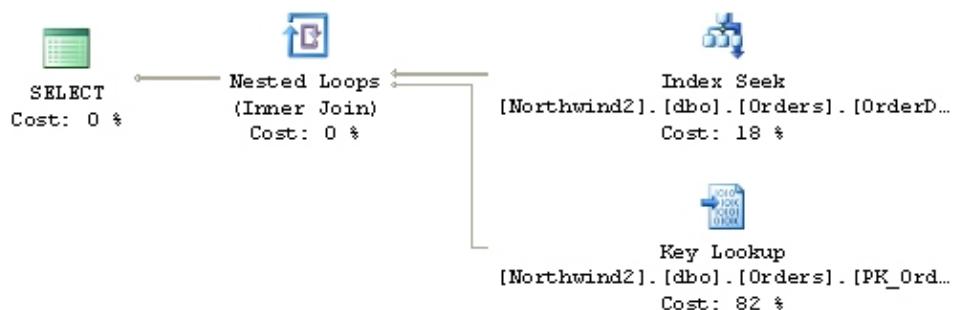
The SQL Server 2005 plans may look different from the SQL Server 2000 plans, but logically they are identical. You can tell that a clustered index seek is a bookmark lookup by the LOOKUP keyword in text plans or by the attribute Lookup="1" in XML plans. For example, here is the text plan for the previous query executed on SQL Server 2005:

Code View:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
  |--Index Seek(OBJECT:([Orders].[OrderDate]),
    SEEK:([Orders].[OrderDate]='1998-02-26') ORDERED FORWARD)
  |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders])),
    SEEK:([Orders].[OrderID]=[Orders].[OrderID]) ORDERED FORWARD)
```

In SQL Server 2005 and SQL Server 2005 SP1, a bookmark lookup in graphical plans uses the same icon as any other clustered index seek. We can only distinguish a normal clustered index seek from a bookmark lookup by checking for the Lookup property. In SQL Server 2005 SP2, a bookmark lookup in a graphical plan uses a new Key Lookup icon. This new icon makes the distinction between a normal clustered index seek and a bookmark lookup very clear. However, note that internally there was no change to the operator between SP1 and SP2. [Figure 3-10](#) illustrates the graphical plan in SQL Server 2005. If you're used to looking at SQL Server 2000 plans, you might find it hard to get used to the representation in SQL Server 2005, but as mentioned previously, logically SQL Server is still doing the same work. You might eventually find SQL Server 2005's representation more enlightening, as it makes it clearer that SQL Server is performing multiple lookups into the underlying table.

Figure 3-10. Graphical plan for index seek and bookmark lookup in SQL Server 2005 SP2



Bookmark lookup can be used with heaps as well as with clustered indexes, as shown above. In SQL Server 2000, a bookmark lookup on a heap looks the same as a bookmark lookup on a clustered index. In SQL Server 2005, a bookmark lookup on a heap still uses a nested loops join, but instead of a clustered index seek, SQL Server uses a RID lookup operator. A RID lookup operator includes a seek predicate on the heap bookmark, but a heap is not an index and a RID lookup is not an index seek.

Bookmark lookup is not a cheap operation. Assuming (as is commonly the case) that no correlation exists between the nonclustered and clustered index keys, each bookmark lookup performs a random I/O into the clustered index or heap. Random I/Os are very expensive. When comparing various plan alternatives including scans, seeks, and seeks with bookmark lookups, the optimizer must decide whether it is cheaper to

perform more sequential I/Os and touch more rows using an index scan (or an index seek with a less selective predicate) that covers all required columns, or to perform fewer random I/Os and touch fewer rows using a seek with a more selective predicate and a bookmark lookup. Because random I/Os are so much more expensive than sequential I/Os, the cutoff point beyond which a clustered index scan becomes cheaper than an index seek with a bookmark lookup generally involves a surprisingly small percentage of the total table—often just a few percent of the total rows.

Tip



In some cases, you can introduce a better plan option by creating a new index or by adding one or more columns to an existing index so as to eliminate a bookmark lookup or change a scan into a seek. In SQL Server 2000, the only way to add columns to an index is to add additional key columns. As noted previously, in SQL Server 2005, you can also add columns using the INCLUDE clause of the CREATE INDEX statement. Included columns are more efficient than key columns. Compared to adding an extra key column, adding an included column uses less disk space and makes searching and updating the index more efficient. Of course, whenever you create new indexes or add new keys or included columns to an existing index, you do consume additional disk space and you do make it more expensive to search and update the index. Thus, you must balance the frequency and importance of the queries that benefit from the new index against the queries or updates that are slower.

Joins

SQL Server supports three physical join operators: nested loops join, merge join, and hash join. We've already seen a nested loops join in the bookmark lookup example. In the following sections, we take a detailed look at how each of these join operators works, explain what logical join types each operator supports, and discuss the performance trade-offs of each join type.

Before we get started, let's put one common myth to rest. There is no "best" join operator, and no join operator is inherently good or bad. We cannot draw any conclusions about a query plan merely from the presence of a particular join operator. Each join operator performs well in the right circumstances and poorly in the wrong circumstances. As we describe each join operator, we will discuss its strengths and weaknesses and the conditions and circumstances under which it performs well.

Nested Loops Join

The nested loops join is the simplest and most basic join algorithm. It compares each row from one table (known as the "outer table") to each row from the other table (known as the "inner table"), looking for rows that satisfy the join predicate.

Note



The terms inner and outer are overloaded; we must infer their meaning from context. "Inner table" and "outer table" refer to the inputs to the join. "Inner join" and "outer join" refer to the semantics of the logical join operations.

We can express the nested loops join algorithm in pseudo-code as:

```

for each row R1 in the outer table
    for each row R2 in the inner table
        if R1 joins with R2
            return (R1, R2)

```

It's the nesting of the loops in this algorithm that gives nested loops join its name.

The total number of rows compared and, thus, the cost of this algorithm is proportional to the size of the outer table multiplied by the size of the inner table. Since this cost grows quickly as the size of the input tables grow, in practice the optimizer tries to minimize the cost by reducing the number of inner rows that must be processed for each outer row.

For example, consider this query:

```

SELECT O.[OrderId]
FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London'

```

When we execute this query, we get the following query plan:

Code View:

```

Rows Executes
46 1      |--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
6   1          |--Index Seek(OBJECT:([Customers].[City] AS [C]),
                           SEEK:([C].[City]=N'London') ORDERED FORWARD)
46 6          |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
                           SEEK:(([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)

```

Unlike most of the examples in this chapter, this plan was generated using SET STATISTICS PROFILE ON so that we can see the number of rows and executions for each operator. The outer table in this plan is Customers while the inner table is Orders. Thus, according to the nested loops join algorithm, SQL Server begins by seeking on the Customers table. The join takes one customer at a time and, for each customer, it performs an index seek on the Orders table. Since there are six customers, it executes the index seek on the Orders table six times. Notice that the index seek on the Orders table depends on the CustomerId, which comes from the Customers table. Each of the six times that SQL Server repeats the index seek on the Orders table, CustomerId has a different value. Thus, each of the six executions of the index seek is different and returns different rows.

We refer to CustomerId as a correlated parameter. If a nested loops join has correlated parameters, they appear in the plan as OUTER REFERENCES. We often refer to this type of nested loops join in which we have an index seek that depends on a correlated parameter as an index join. An index join is possibly the most common type of nested loops join. In fact, in SQL Server 2005, as we've already seen, a bookmark lookup is simply an index join between a nonclustered index and the base table.

The prior example illustrated two important techniques that SQL Server uses to boost the performance of a nested loops join: correlated parameters and, more importantly, an index seek based on those correlated parameters on the inner side of the join. Another performance optimization that we don't see here is the use of a lazy spool on the inner side of the join. A lazy spool caches and can reaccess the results from the inner side of the join. A lazy spool is especially useful when there are correlated parameters with many duplicate values

and when the inner side of the join is relatively expensive to evaluate. By using a lazy spool, SQL Server can avoid recomputing the inner side of the join multiple times with the same correlated parameters. We will see some examples of spools including lazy spools later in this chapter.

Not all nested loops joins have correlated parameters. A simple way to get a nested loops join without correlated parameters is with a cross join. A cross join matches all rows of one table with all rows of the other table. To implement a cross join with a nested loops join, we must scan and join every row of the inner table to every row of the outer table. The set of inner table rows does not change depending on which outer table row we are processing. Thus, with a cross join, there can be no correlated parameter.

In some cases, if we do not have a suitable index or if we do not have a join predicate that is suitable for an index seek, the optimizer may generate a query plan without correlated parameters. The rules for determining whether a join predicate is suitable for use with an index seek are identical to the rules for determining whether any other predicate is suitable for an index seek. For example, consider the following query, which returns the number of employees who were hired after each other employee:

```
SELECT E1.[EmployeeId], COUNT(*)
FROM [Employees] E1 JOIN [Employees] E2
    ON E1.[HireDate] < E2.[HireDate]
GROUP BY E1.[EmployeeId]
```

We have no index on the HireDate column. Thus, this query generates a simple nested loops join with a predicate but without any correlated parameters and without an index seek:

Code View:

```
|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
|--Stream Aggregate(GROUP BY:([E1].[EmployeeID]) DEFINE:([Expr1007]=Count(*)))
|--Nested Loops(Inner Join, WHERE:([E1].[HireDate]<[E2].[HireDate]))
|--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
|--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E2]))
```

Do not be concerned with the aggregation. The purpose of this example is to illustrate the behavior of the nested loops join. We will discuss aggregation later in this chapter.

Now consider the following identical query which has been rewritten to use a CROSS APPLY:

```
SELECT E1.[EmployeeId], ECnt.Cnt
FROM [Employees] E1 CROSS APPLY
(
    SELECT COUNT(*) Cnt
    FROM [Employees] E2
    WHERE E1.[HireDate] < E2.[HireDate]
) ECnt
```

Although these two queries are identical, and will always return the same results, the plan for the query with the CROSS APPLY uses a nested loops join with a correlated parameter:

Code View:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([E1].[HireDate]))
|--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
```

```

|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
|--Stream Aggregate(DEFINE:([Expr1007]=Count(*)))
|--Clustered Index Scan (OBJECT:([Employees].[PK_Employees] AS [E2]),
    WHERE:([E1].[HireDate]<[E2].[HireDate]))

```

Tip



This example demonstrates that, in some cases, small changes to a query, even rewrites that do not change the semantics of the query, can yield substantially different query plans. In particular, in some cases, a CROSS APPLY can induce the optimizer to generate a correlated nested loops join, which may or may not be desirable.

Join Predicates and Logical Join Types

The nested loops join is one of the most flexible join methods. It supports all join predicates including equijoin (equality) predicates and inequality predicates.

The nested loops join supports the following logical join operators:

- Inner join
- Left outer join
- Cross join
- Cross apply and outer apply
- Left semi-join and left anti-semi-join

The nested loops join does not support the following logical join operators:

- Right and full outer join
- Right semi-join and right anti-semi-join

To understand why the nested loops join does not support right outer or semi-joins, let's first look at how we can extend the nested loops join algorithm to support left outer and semi-joins. Here is the pseudo-code for a left outer join. We need only two extra lines of pseudo-code to extend the inner join algorithm.

```

for each row R1 in the outer table
begin
    for each row R2 in the inner table
        if R1 joins with R2
            output (R1, R2)

end

```

This algorithm keeps track of whether we joined a particular outer row. If after exhausting all inner rows, we find that a particular inner row did not join, we output it as a NULL extended row. We can write similar pseudo-code for a left semi-join or left anti-semi-join. [A semi-join or anti-semi-join returns one half of the input information, that is, columns from one of the joined tables. So instead of outputting (R1, R2) as in the pseudo-code above, a left semi-join outputs just R1. Moreover, a semi-join returns each row of the outer table

at most once. Thus, after finding a match and outputting a given row R1, a left semi-join moves immediately to the next outer row. A left anti-semi-join returns a row from R1 if it does not match with R2.]

Now consider how we might support right outer join. In this case, we want to return pairs (R1, R2) for rows that join and pairs (NULL, R2) for rows of the inner table that do not join. The problem is that we scan the inner table multiple times—once for each row of the outer join. We may encounter the same inner rows multiple times during these multiple scans. At what point can we conclude that a particular inner row has not or will not join? Moreover, if we are using an index join, we might not encounter some inner rows at all. Yet these rows should also be returned for an outer join. Further analysis uncovers similar problems for right semi-joins and right anti-semi-joins.

Fortunately, since right outer join commutes into left outer join and right semi-join commutes into left semi-join, SQL Server can use the nested loops join for right outer and semi-joins. However, while these transformations are valid, they may affect performance. When the optimizer transforms a right join into a left join, it also switches the outer and inner inputs to the join. Recall that to use an index join, the index needs to be on the inner table. By switching the outer and inner inputs to the table, the optimizer also switches the table on which we need an index to be able to use an index join.

Full Outer Joins

The nested loops join cannot directly support full outer join. However, the optimizer can transform [Table1] FULL OUTER JOIN [Table2] into [Table1] LEFT OUTER JOIN [Table2] UNION ALL [Table2] LEFT ANTI-SEMI-JOIN [Table1]. Basically, this transforms the full outer join into a left outer join—which includes all pairs of rows from Table1 and Table2 that join and all rows of Table1 that do not join—then adds back the rows of Table2 that do not join using an anti-semi-join. To demonstrate this transformation, suppose that we have two customer tables. Further suppose that each customer table has different customer ids. We want to merge the two lists while keeping track of the customer ids from each table. We want the result to include all customers regardless of whether a customer appears in both lists or in just one list. We can generate this result with a full outer join. We'll make the rather unrealistic assumption that two customers with the same name are indeed the same customer.

Code View:

```
CREATE TABLE [Customer1] ([CustomerId] int PRIMARY KEY, [Name] nvarchar(30))
CREATE TABLE [Customer2] ([CustomerId] int PRIMARY KEY, [Name] nvarchar(30))

SELECT C1.[Name], C1.[CustomerId], C2.[CustomerId]
FROM [Customer1] C1 FULL OUTER JOIN [Customer2] C2
    ON C1.[Name] = C2.[Name]
```

Here is the plan for this query, which demonstrates the transformation in action:

Code View:

```
--Concatenation
|--Nested Loops(Left Outer Join, WHERE:([C1].[Name]=[C2].[Name]))
|   |--Clustered Index Scan(OBJECT:([Customer1].[PK_Customer1] AS [C1]))
|   |--Clustered Index Scan(OBJECT:([Customer2].[PK_Customer2] AS [C2]))
|--Compute Scalar(DEFINE:([C1].[CustomerId]=NULL, [C1].[Name]=NULL))
   |--Nested Loops(Left Anti Semi Join, WHERE:([C1].[Name]=[C2].[Name]))
      |--Clustered Index Scan(OBJECT:([Customer2].[PK_Customer2] AS [C2]))
      |--Clustered Index Scan(OBJECT:([Customer1].[PK_Customer1] AS [C1]))
```

The concatenation operator implements the UNION ALL. We'll cover this operator in a bit more detail when we discuss unions later in this chapter.

Costing

The complexity or cost of a nested loops join is proportional to the size of the outer input multiplied by the size of the inner input. Thus, a nested loops join generally performs best for relatively small input sets. The inner input need not be small, but, if it is large, it helps to include an index on a highly selective join key.

In some cases, a nested loops join is the only join algorithm that SQL Server can use. SQL Server must use a nested loops join for cross join as well as for some complex cross applies and outer applies. Moreover, as we are about to see, with one exception, a nested loops join is the only join algorithm that SQL Server can use without at least one equijoin predicate. In these cases, the optimizer must choose a nested loops join regardless of cost.

Note



Merge join supports full outer joins without an equijoin predicate. We will discuss this unusual scenario in the next section.

Partitioned Tables

In SQL Server 2005, the nested loops join is also used to implement query plans that scan partitioned tables. To see an example of this use of the nested loops join, we need to create a partitioned table. The following script creates a simple partition function and scheme that defines four partitions, creates a partitioned table using this scheme, and then selects rows from the table:

Code View:

```
CREATE PARTITION FUNCTION [PtnFn] (int) AS RANGE FOR VALUES (1, 10, 100)
CREATE PARTITION SCHEME [PtnSch] AS PARTITION [PtnFn] ALL TO ([PRIMARY])
CREATE TABLE [PtnTable] ([PK] int PRIMARY KEY, [Data] int) ON [PtnSch] ([PK])

SELECT [PK], [Data] FROM [PtnTable]
```

SQL Server assigns sequential partition ids to each of the four partitions defined by the partition scheme. The range for each partition is shown in [Table 3-4](#).

Table 3-4. The Range of Values in Each of Our Four Partitions

PartitionID **Values**
1 [PK] <= 121 < [PK] <= 103
10 [PK] <= 103 < [PK] <= 1004
100 [PK] <= 1004 < [PK]

The query plan for the SELECT statement uses a constant scan operator to enumerate these four partition ids and a special nested loops join to execute a clustered index scan of each of these four partitions:

Code View:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([PtnIds1003])
 |--  
 |--Clustered Index Scan(OBJECT:([PtnTable].[PK_PtnTable]))
```

Observe that the nested loops join explicitly identifies the partition id column as [PtnIds1003]. Although it is not obvious from the text plan, the clustered index scan uses the partition id column and checks it on each execution to determine which partition to scan. This information is clearly visible in XML plans:

Code View:

```
<RelOp NodeId="6" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index Scan" ...>
  <IndexScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
    <Object ... Table="[PtnTable]" Index="[PK_PtnTable]" />

  </IndexScan>
</RelOp>
```

Merge Join

Now let's look at merge join. Unlike the nested loops join, which supports any join predicate, the merge join requires at least one equijoin predicate. Moreover, the inputs to the merge join must be sorted on the join keys. For example, if we have a join predicate [Customers].[CustomerId] = [Orders].[CustomerId], the Customers and Orders tables must both be sorted on the CustomerId column.

The merge join works by simultaneously reading and comparing the two sorted inputs one row at a time. At each step, it compares the next row from each input. If the rows are equal, it outputs a joined row and continues. If the rows are not equal, it discards the lesser of the two inputs and continues. Since the inputs are sorted, any row that the join discards must be less than any of the remaining rows in either input and, thus, can never join. A merge join does not necessarily need to scan every row from both inputs. As soon as it reaches the end of either input, the merge join stops scanning.

We can express the algorithm in pseudo-code as:

```
get first row R1 from input 1
get first row R2 from input 2
while not at the end of either input
begin
  if R1 joins with R2
  begin
    output (R1, R2)
    get next row R2 from input 2
  end
  else if R1 < R2
    get next row R1 from input 1
  else
    get next row R2 from input 2
end
```

Unlike the nested loops join where the total cost may be proportional to the product of the number of rows in the input tables, with a merge join each table is read at most once and the total cost is proportional to the sum of the number of rows in the inputs. Thus, merge join is often a better choice for larger inputs.

One-to-Many vs. Many-to-Many Merge Join

The above pseudo-code implements a one-to-many merge join. After it joins two rows, it discards R2 and moves to the next row of input 2. This presumes that it will never find another row from input 1 that will ever join with the discarded row. In other words, there can't be duplicates in input 1. On the other hand, it is acceptable that there might be duplicates in input 2 since it did not discard the current row from input 1.

Merge join can also support many-to-many merge joins. In this case, it must keep a copy of each row from input 2 whenever it joins two rows. This way, if it later finds a duplicate row from input 1, it can play back the saved rows. On the other hand, if it finds that the next row from input 1 is not a duplicate, it can discard the saved rows. The merge join saves these rows in a worktable in tempdb. The amount of required disk space depends on the number of duplicates in input 2.

A one-to-many merge join is always more efficient than a many-to-many merge join since it does not need a worktable. To use a one-to-many merge join, the optimizer must be able to determine that one of the inputs consists strictly of unique rows. Typically, this means that either there is a unique index on the input or there is an explicit operator in the plan (perhaps a sort distinct or a group by) to ensure that the input rows are unique.

Sort Merge Join vs. Index Merge Join

There are two ways that SQL Server can get sorted inputs for a merge join: It may explicitly sort the inputs using a sort operator, or it may read the rows from an index. In general, a plan using an index to achieve sort order is cheaper than a plan using an explicit sort.

Join Predicates and Logical Join Types

Merge join supports multiple equijoin predicates as long as the inputs are sorted on all of the join keys. The specific sort order does not matter as long as both inputs are sorted in the same order. For example, if we have a join predicate $T1.[Col1] = T2.[Col1]$ and $T1.[Col2] = T2.[Col2]$, we can use a merge join as long as tables T1 and T2 are both sorted either on (Col1, Col2) or on (Col2, Col1).

Merge join also supports residual predicates. For example, consider the join predicate $T1.[Col1] = T2.[Col1]$ and $T1.[Col2] > T2.[Col2]$. Although the inequality predicate cannot be used as part of a merge join, the equijoin portion of this predicate can be used to perform a merge join (presuming both tables are sorted on [Col1]). For each pair of rows that joins on the equality portion of predicate, the merge join can then apply the inequality predicate. If the inequality evaluates to true, the join returns the row; if not, it discards the row.

Merge join supports all outer and semi-join variations. For instance, to implement an outer join, the merge join simply needs to track whether each row has joined. Instead of discarding a row that has not joined, it can NULL extend it and output it as appropriate. Note that, unlike the inner join case where a merge join can stop as soon as it reaches the end of either input, for an outer (or anti-semi-) join the merge join must scan to the end of whichever input it is preserving. For a full outer join, it must scan to the end of both inputs.

Merge join supports a special case for full outer join. In some cases, the optimizer generates a merge join for a full outer join even if there is no equijoin predicate. This join is equivalent to a many-to-many merge join where all rows from one input join with all rows from the other input. As with any other many-to-many merge join, SQL Server builds a worktable to store and play back all rows from the second input. SQL Server

supports this plan as an alternative to the previously discussed transformation used to support full outer join with nested loops join.

Examples

Because merge join requires that input rows be sorted, the optimizer is most likely to choose a merge join when we have an index that returns rows in that sort order. For example, the following query simply joins the Orders and Customers tables:

```
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]
```

Since we have no predicates other than the join predicates, we must scan both tables in their entirety. Moreover, we have covering indexes on the CustomerId column of both tables. Thus, the optimizer chooses a merge join plan:

Code View:

```
--Merge Join(Inner Join, MERGE:([C].[CustomerID])=([O].[CustomerID]), RESIDUAL:(...))
--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED FORWARD)
--Index Scan(OBJECT:([Orders].[CustomerID] AS [O]), ORDERED FORWARD)
```

Observe that this join is one to many. We can tell that it is one to many by the absence of the MANY-TO-MANY keyword in the query plan. We have a unique index (actually a primary key) on the CustomerId column of the Customers table. Thus, the optimizer knows that there will be no duplicate CustomerId values from this table and chooses the one-to-many join.

Note that for a unique index to enable a one-to-many join, we must be joining on all of the key columns of the unique index. It is not sufficient to join on a subset of the key columns as the index only guarantees uniqueness on the entire set of key columns.

Now let's consider a slightly more complex example. The following query returns a list of orders that shipped to cities different from the city that we have on file for the customer who placed the order:

```
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId] AND O.[ShipCity] <> C.[City]
ORDER BY C.[CustomerId]
```

We need the ORDER BY clause to encourage the optimizer to choose a merge join. We'll return to this point in a moment. Here is the query plan:

Code View:

```
--Merge Join(Inner Join, MERGE:([C].[CustomerID])=([O].[CustomerID]),
RESIDUAL:(... AND [O].[ShipCity]<>[C].[City]))
--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED FORWARD)
--Sort(ORDER BY:([O].[CustomerID] ASC))
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

There are a couple of points worth noting about this new plan. First, because this query needs the ShipCity column from the Orders table for the extra predicate, the optimizer cannot use a scan of the CustomerId index, which does not cover the extra column, to get rows from the Orders table sorted by the CustomerId column. Instead, the optimizer chooses to scan the clustered index and sort the results. The ORDER BY clause requires that the optimizer add this sort either before the join, as in this example, or after the join. By performing the sort before the join, the plan can take advantage of the merge join. Moreover, the merge join preserves the input order so there is no need to sort the data again after the join.

Note



Technically, the optimizer could decide to use a scan of the CustomerId index along with a bookmark lookup, but since it is scanning the entire table, the bookmark lookup would be prohibitively expensive.

Second, this merge join demonstrates a residual predicate: O.[ShipCity] <> C.[City]. The optimizer cannot use this predicate as part of the join's merge keys because it is an inequality. However, as the example shows, as long as there is at least one equality predicate, SQL Server can use the merge join.

Hash Join

Hash join is the third physical join operator. When it comes to physical join operators, hash join does the heavy lifting. While nested loops join works well with relatively small data sets and merge join helps with moderately-sized data sets, hash join excels at performing the largest joins. Hash joins parallelize and scale better than any other join and are great at minimizing response times for data warehouse queries.

Hash join shares many characteristics with merge join. Like merge join, it requires at least one equijoin predicate, supports residual predicates, and supports all outer and semi-joins. Unlike merge join, it does not require ordered input sets and, while it does support full outer join, it does require an equijoin predicate.

The hash join algorithm executes in two phases known as the "build" and "probe" phases. During the build phase, it reads all rows from the first input (often called the left or build input), hashes the rows on the equijoin keys, and creates or builds an in-memory hash table. During the probe phase, it reads all rows from the second input (often called the right or probe input), hashes these rows on the same equijoin keys, and looks or probes for matching rows in the hash table. Since hash functions can lead to collisions (two different key values that hash to the same value), the hash join typically must check each potential match to ensure that it really joins. Here is pseudo-code for this algorithm:

```

for each row R1 in the build table
begin
    calculate hash value on R1 join key(s)
    insert R1 into the appropriate hash bucket
end
for each row R2 in the probe table
begin
    calculate hash value on R2 join key(s)
    for each row R1 in the corresponding hash bucket
        if R1 joins with R2
            output (R1, R2)
end

```

Note that unlike the nested loops and merge joins, which immediately begin flowing output rows, the hash join is blocking on its build input. That is, it must read and process its entire build input before it can return any rows. Moreover, unlike the other join methods, the hash join requires a memory grant to store the hash table. Thus, there is a limit to the number of concurrent hash joins that SQL Server can run at any given time. While these characteristics and restrictions are generally not a problem for data warehouses, they are undesirable for most OLTP applications.

Note



A sort merge join does require a memory grant for the sort operator(s) but does not require a memory grant for the merge join itself.

Memory and Spilling

Before a hash join begins execution, SQL Server tries to estimate how much memory it will need to build its hash table. It uses the cardinality estimate for the size of the build input along with the expected average row size to estimate the memory requirement. To minimize the memory required by the hash join, the optimizer chooses the smaller of the two tables as the build table. SQL Server then tries to reserve sufficient memory to ensure that the hash join can successfully store the entire build table in memory.

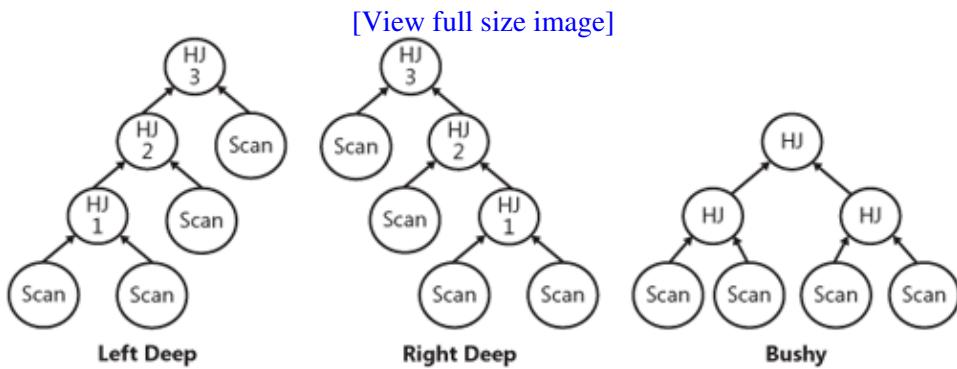
What happens if SQL Server grants the hash join less memory than it requests or if the estimate is too low? In these cases, the hash join may run out of memory during the build phase. If the hash join runs out of memory, it begins spilling a small percentage of the total hash table to disk (to a workfile in tempdb). The hash join keeps track of which buckets of the hash table are still in memory and which ones have been spilled to disk. As it reads each new row from the build table, it checks to see whether it hashes to an in-memory or an on-disk bucket. If it hashes to an in-memory bucket, it proceeds normally. If it hashes to an on-disk bucket, it writes the row to disk. This process of running out of memory and spilling buckets to disk may repeat multiple times until the build phase is complete.

The hash join performs a similar process during the probe phase. For each new row from the probe table, it checks to see whether it hashes to an in-memory or an on-disk bucket. If it hashes to an in-memory bucket, it probes the hash table, produces any appropriate joined rows, and discards the row. If it hashes to an on-disk bucket, it writes the row to disk. Once the join completes the first pass of the probe table, it returns one by one to any buckets that spilled, reads the build rows back into memory, reconstructs the hash table for each bucket, and then reads the corresponding probe bucket and completes the join. If while processing spilled sets of buckets, the hash join again runs out of memory, the process simply repeats. We refer to the number of times that the hash join repeats this algorithm and spills the same data as the "recursion level." After a set number of recursion levels, if the hash join continues to spill, it switches to a special "bailout" algorithm that, while less efficient, is guaranteed to complete eventually.

Left Deep vs. Right Deep vs. Bushy Hash Join Trees

The shape and order of joins in a query plan can significantly impact the performance of the plan. The shape of a query plan is so important that we actually have terms for the most common shapes. The termsâ€”left deep, right deep, and bushyâ€”are based on the physical appearance of the query plan, as illustrated by [Figure 3-11](#).

Figure 3-11. Three common shapes for query plans involving joins



The shape of the join tree is particularly interesting for hash joins as it affects the memory consumption.

In a left deep tree, the output of one hash join is the build input to the next hash join. Because hash joins consume their entire build input before moving to the probe phase, in a left deep tree only adjacent pairs of hash joins are active at the same time. For example, for the left deep example in Figure 3-11, SQL Server begins by building the hash table for HJ1. When HJ1 begins probing, HJ2 begins building its hash table. When HJ1 is done probing, SQL Server can release the memory used by its hash table. Only then does HJ2 begin probing and HJ3 begin building its hash table. Thus, HJ1 and HJ3 are never active at the same time and can share the same memory grant. The total memory requirement is the maximum of the memory needed by any two adjacent joins (that is, HJ1 and HJ2 or HJ2 and HJ3).

In a right deep tree, the output of one hash join is the probe input to the next hash join. All of the hash joins build their complete hash tables before any begin the probe phase of the join. All of the hash joins are active at once and cannot share memory. When SQL Server does begin the probe phase of the join, the rows flow up the entire tree of hash joins without blocking. Thus, the total memory requirement is the sum of the memory needed by all three joins.

Examples

The following query is nearly identical to the earlier merge join example except that we select one additional column, the OrderDate column, from the Orders table:

```
SELECT O.[OrderId], O.[OrderDate], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
    ON O.[CustomerId] = C.[CustomerId]
```

Because the CustomerId index on the Orders table does not cover the OrderDate column, we would need a sort to use a merge join. We saw this outcome in the second merge join example, but this time we do not have an ORDER BY clause. Thus, the optimizer chooses the following hash join plan:

Code View:

```
--Hash Match(Inner Join, HASH:([C].[CustomerID])=([O].[CustomerID]), RESIDUAL:(...))
--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]))
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

Summary of Join Properties

[Table 3-5](#) summarizes the characteristics of the three physical join operators.

Table 3-5. Characteristics of the Three Join Algorithms

Nested Loops Join	Relatively small inputs with an index on the inner table on the join key.
Merge Join	Medium to large inputs with indexes to provide order on the equijoin keys and/or where we require order after the join.
Hash Join	Data warehouse queries with medium to large inputs. Scalable parallel execution.
Concurrency	Supports large numbers of concurrent users.
Many-to-one join with order provided by indexes (rather than explicit sorts)	Supports large numbers of concurrent users.
Best for small numbers of concurrent users.	
Stop and go	No
Yes	Yes (build input only)
Outer and semi-joins	Equijoin required
Left joins only (full outer joins via transformation)	No (except for full outer join)
All join types	All join types
Uses memory	No
(may require sorts which use memory)	Yes
Yes	Uses tempdb
Requires order	No
Yes	Yes (many-to-many join only)
Yes	Yes (if join runs out of memory and spills)
Yes	Requires order
Yes	Preserves order
Only	Yes
Yes	Outer input
Yes	Yes
Yes	Supports dynamic cursors
Yes	Yes
No	No

Aggregations

SQL Server supports two physical operators for performing aggregations. These operators are stream aggregate and hash aggregate.

Scalar Aggregation

Scalar aggregates are queries with aggregate functions in the select list and no GROUP BY clause. Scalar aggregates always return a single row. SQL Server always implements scalar aggregates using the stream aggregate operator.

Let's begin by considering a trivial example:

```
SELECT COUNT(*) FROM [Orders]
```

This query produces the following plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))
 |--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))
 |--Index Scan(OBJECT:([Orders].[ShippersOrders]))
```

The stream aggregate operator just counts the number of input rows and returns this result. The stream aggregate actually computes the count ([Expr1004]) as a BIGINT. The compute scalar is needed to convert this result to the expected output type of INT. Note that a scalar stream aggregate is one of the only examples of a nonleaf operator that can produce an output row even with an empty input set.

It is easy to see how to implement other simple scalar aggregate functions such as MIN, MAX, and SUM. A single-stream aggregate operator can calculate multiple scalar aggregates at the same time:

```
SELECT MIN([OrderDate]), MAX([OrderDate]) FROM [Orders]
```

Here is the query plan with a single-stream aggregate operator:

```
|--Stream Aggregate(DEFINE:([Expr1003]=MIN([Orders].[OrderDate]),
[Expr1004]=MAX([Orders].[OrderDate])))
|--Index Scan(OBJECT:([Orders].[OrderDate]))
```

Note that SQL Server does not need to convert the result for the MIN and MAX aggregates since the data types of these aggregates are computed based on the data type of the OrderDate column.

Some aggregates such as AVG are actually calculated from two other aggregates such as SUM and COUNT:

```
SELECT AVG([Freight]) FROM [Orders]
```

Notice how the compute scalar operator in the plan computes the average from the sum and count:

```
|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
THEN NULL
ELSE [Expr1005]/CONVERT_IMPLICIT(money,[Expr1004],0)
END))
|--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
[Expr1005]=SUM([Orders].[Freight])))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

The CASE expression is needed to make sure that SQL Server does not attempt to divide by zero.

Although SUM does not need to be computed per se, it still needs the count:

```
SELECT SUM([Freight]) FROM [Orders]
```

Notice how the CASE expression in this query plan uses the COUNT to ensure that SUM returns NULL instead of zero if there are no rows:

```
|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
THEN NULL
ELSE [Expr1005]
END))
|--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
[Expr1005]=SUM([Orders].[Freight])))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Scalar Distinct

Now let's take a look at what happens if we add a DISTINCT keyword to a scalar aggregate. Consider this query to compute the number of distinct cities to which we've shipped orders:

```
SELECT COUNT(DISTINCT [ShipCity]) FROM [Orders]
```

This query produces this query plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(DEFINE:([Expr1006]=COUNT([Orders].[ShipCity])))
    |--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
        |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Since the query must only count rows that have a unique value for the ShipCity column, SQL Server adds a sort distinct operator to eliminate rows with duplicate ShipCity values. Sort distinct is one of the common methods used by SQL Server to eliminate duplicates. It is easy to remove duplicate rows after sorting the input set since the duplicates are then adjacent to one another. There are other methods that SQL Server can employ to eliminate duplicates, as we'll see shortly. Other than the addition of the sort operator, this plan is the same as the COUNT(*) plan with which we began our discussion of aggregation.

Not all distinct aggregates require duplicate elimination. For example, MIN and MAX behave identically with and without the distinct keyword. The minimum and maximum values of a set remain the same whether or not the set includes duplicate values. For example, this query gets the same plan as the above MIN/MAX query without the DISTINCT keyword.

```
SELECT MIN(DISTINCT [OrderDate]), MAX(DISTINCT [OrderDate]) FROM [Orders]
```

If we have a unique index, SQL Server also can skip the duplicate elimination because the index guarantees that there are no duplicates. For example, the following query is identical to the simple COUNT(*) query with which we began this discussion:

```
SELECT COUNT(DISTINCT [OrderId]) FROM [Orders]
```

Multiple Distinct

Consider this query:

```
SELECT COUNT(DISTINCT [ShipAddress]), COUNT(DISTINCT [ShipCity])
FROM [Orders]
```

As we've seen, SQL Server can compute COUNT(DISTINCT [ShipAddress]) by eliminating rows that have duplicate values for the ShipAddress column. Similarly, SQL Server can compute COUNT(DISTINCT [ShipCity]) by eliminating rows that have duplicate values for the ShipCity column. But, given that these two sets of rows are different, how can SQL Server compute both at the same time? The answer is it cannot. It

must first compute one aggregate result, then the other, and then it must combine the two results into a single output row:

Code View:

```
|--Nested Loops(Inner Join)
|  |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1009],0)))
|  |  |--Stream Aggregate(DEFINE:([Expr1009]=COUNT([Orders].[ShipAddress])))
|  |  |  |--Sort(DISTINCT ORDER BY:([Orders].[ShipAddress] ASC))
|  |  |  |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
|  |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1010],0)))
|  |  |--Stream Aggregate(DEFINE:([Expr1010]=COUNT([Orders].[ShipCity])))
|  |  |  |--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
|  |  |  |--Clustered Index Scan
|  |  |  (OBJECT:([Orders].[PK_Orders]))
```

The two inputs to the nested loops join compute the two counts from the original query. One of the inputs removes duplicates and computes the count for the ShipAddress column, while the other input removes duplicates and computes the count for the ShipCity column. The nested loops join has no join predicate; it is a cross join. Since both inputs to the nested loops join each produce a single row— they are both scalar aggregates—the result of the cross join is also a single row. The cross join just serves to "glue" the two columns of the result into a single row.

If we have more than two distinct aggregates on different columns, the optimizer just uses more than one cross join. The optimizer also uses this type of plan for a mix of nondistinct and distinct aggregates. In that case, the optimizer uses a single cross join input to calculate all of the nondistinct aggregates.

Stream Aggregation

Now that we've seen how to compute scalar aggregates, let's see how SQL Server computes general purpose aggregates where we have a GROUP BY clause. We'll begin by taking a closer look at how stream aggregation works.

The Algorithm

Stream aggregate relies on data arriving sorted by the GROUP BY column(s). Like a merge join, if a query includes a GROUP BY clause with more than one column, the stream aggregate can use any sort order that includes all of the columns. For example, a stream aggregate can group on columns Col1 and Col2 with data sorted on (Col1, Col2) or on (Col2, Col1). As with merge join, the sort order may be delivered by an index or by an explicit sort operator. The sort order ensures that sets of rows with the same value for the GROUP BY columns will be adjacent to one another.

Here is pseudo-code for the stream aggregate algorithm:

```
clear the current aggregate results
clear the current group by columns
for each input row
begin
    if the input row does not match the current group by columns
    begin
        output the current aggregate results (if any)
        clear the current aggregate results
        set the current group by columns to the input row
    end
```

```
update the aggregate results with the input row
end
```

For example, to compute a SUM, the stream aggregate considers each input row. If the input row belongs to the current group (that is, the group by columns of the input row match the group by columns of the previous row), the stream aggregate updates the current SUM by adding the appropriate value from the input row to the running total. If the input row belongs to a new group (that is, the group by columns of the input row do not match the group by columns of the previous row), the stream aggregate outputs the current SUM, resets the SUM to zero, and starts a new group.

Simple Examples

Consider the following query that counts the number of orders shipped to each address:

```
SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]
```

Here is the plan for this query:

Code View:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
|--Stream Aggregate(GROUP BY: ([Orders].[ShipCity], [Orders].[ShipAddress])
    DEFINE:([Expr1006]=Count(*)))
|--Sort(ORDER BY:([Orders].[ShipCity] ASC, [Orders].[ShipAddress] ASC))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

This plan is basically the same as the one that we saw for the scalar aggregate queries, except that SQL Server needs to sort the data before it aggregates. We can think of the scalar aggregate as one big group containing all of the rows; thus, for a scalar aggregate there is no need to sort the rows into different groups.

Stream aggregate preserves the input sort order. For example, suppose that we extended the above query with an ORDER BY clause that sorts the results on the GROUP BY keys:

```
SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]
ORDER BY [ShipAddress], [ShipCity]
```

The resulting plan still has only the one sort operator (below the stream aggregate):

Code View:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
|--Stream Aggregate(GROUP BY:([Orders].[ShipAddress],[Orders].[ShipCity])
    DEFINE:([Expr1006]=Count(*)))
|--Sort()
    |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

However, note that the sort columns are reversed from the first example. Previously, it did not matter whether SQL Server sorted on the ShipAddress or ShipCity column first. Now since the query includes an ORDER BY clause, the optimizer chooses to sort on the ShipAddress column first to avoid needing a second sort operator following the stream aggregate.

If we have an appropriate index, the plan does not need a sort operator at all. For instance, consider the following query which counts orders by customer (instead of shipping address):

```
SELECT [CustomerId], COUNT(*)
FROM [Orders]
GROUP BY [CustomerId]
```

The new plan uses an index on the CustomerId column to avoid sorting:

Code View:

```
--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
--Stream Aggregate(GROUP BY:([Orders].[CustomerID])) DEFINE:([Expr1006]=Count(*))
--Index Scan(OBJECT:([Orders].[CustomerID])), ORDERED FORWARD
```

Select Distinct

If we have an index to provide order, SQL Server can also use the stream aggregate to implement SELECT DISTINCT. (If we do not have an index to provide order, the optimizer cannot use a stream aggregate without adding a sort. In this case, the optimizer can just let the sort distinct directly; there is no reason to use a stream aggregate as well.) SELECT DISTINCT is essentially the same as GROUP BY on all selected columns with no aggregate functions. For example:

```
SELECT DISTINCT [CustomerId] FROM [Orders]
```

Can also be written as:

```
SELECT [CustomerId] FROM [Orders] GROUP BY [CustomerId]
```

Both queries use the same plan:

```
--Stream Aggregate(GROUP BY:([Orders].[CustomerID]))
--Index Scan(OBJECT:([Orders].[CustomerID])), ORDERED FORWARD
```

Notice that the stream aggregate has a GROUP BY clause, but no defined columns.

Distinct Aggregates

SQL Server implements distinct aggregates for queries with a GROUP BY clause identically to how it implements distinct scalar aggregates. In both cases, a distinct aggregate needs a plan that eliminates duplicates before aggregating. For example, suppose we would like to find the number of distinct customers served by each employee:

```
SELECT [EmployeeId], COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]
```

This query results in the following plan:

Code View:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
 |   DEFINE:([Expr1006]=COUNT([Orders].[CustomerID])))
 |--Sort(DISTINCT ORDER BY:([Orders].[EmployeeID] ASC,[Orders].[CustomerID] ASC))
 |   |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

However, we just saw how SQL Server can use an aggregate to eliminate duplicates. SQL Server can also use an aggregate to implement distinct aggregates without the sort distinct. To see such a plan, we need to create a nonunique two-column index. For example, let's temporarily create an index on the EmployeeId and CustomerId columns of the Orders table:

```
CREATE INDEX [EmployeeCustomer] ON [Orders] (EmployeeId, CustomerId)
```

Now, the plan looks as follows:

Code View:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
 |   DEFINE:([Expr1006]=COUNT([Orders].[CustomerID])))
 |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID], [Orders].[CustomerID]))
 |   |--Index Scan(OBJECT:([Orders].[EmployeeCustomer])), ORDERED FORWARD)
```

Observe how the optimizer has replaced the sort distinct with a stream aggregate. This plan is possible since, as in the SELECT DISTINCT example above, we have an index that provides order on the columns over which we need to eliminate duplicates. Let's drop the temporary index before we continue:

```
DROP INDEX [Orders].[EmployeeCustomer]
```

Multiple Distincts

Finally, let's take a look at how SQL Server implements a mix of nondistinct and distinct aggregates (or multiple distinct aggregates) in a single GROUP BY query. Suppose that we wish to find the total number of orders taken by each employee, as well as the total number of distinct customers served by each employee:

```
SELECT [EmployeeId], COUNT(*), COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]
```

As in the scalar aggregate with multiple distincts example that we saw earlier, SQL Server cannot compute the two aggregates at the same time. Instead, it must compute each aggregate separately. However, unlike the scalar aggregate example, the GROUP BY clause means each aggregate returns multiple rows. Thus, while the scalar aggregate plan used a cross join to glue together two individual rows, this query needs an inner join on EmployeeId (the group by column) to combine the two sets of rows. Here is the query plan:

Code View:

```
--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
|--Merge Join(Inner Join, MANY-TO-MANY
  MERGE:([Orders].[EmployeeID])=([Orders].[EmployeeID]),RESIDUAL:(...))
  |--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
  |   |--Compute Scalar(DEFINE:(([Expr1004]=CONVERT_IMPLICIT(int,[Expr1011],0)))
  |       |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID]))
  |           DEFINE:(([Expr1011]=COUNT([Orders].[CustomerID])))
  |           |--Sort (DISTINCT ORDER BY:([Orders].[EmployeeID] ASC,
  |               [Orders].[CustomerID] ASC))
  |                   |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
  |--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
  |   |--Compute Scalar(DEFINE:(([Expr1003]=CONVERT_IMPLICIT(int,[Expr1012],0)))
  |       |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID]))
  |           DEFINE:(([Expr1012]=Count(*)))
  |           |--Index Scan(OBJECT:([Orders].[EmployeeID]),ORDERED FORWARD)
```

Notice that since the results of the two aggregations are already sorted on the group by column, the plan can use a merge join. You can see the graphical plan for the query in [Figure 3-12](#).

Figure 3-12. Graphical plan showing stream aggregation and a merge join to compute multiple distinct aggregates



Note



The compute scalar operators that appear to define $[EmployeeId] = [EmployeeId]$ are needed for internal purposes and can be disregarded. The merge join ought to be one to

many not many to many since the aggregates ensure uniqueness on the group by column (and join column). This is a minor performance issue not a correctness issue.

Hash Aggregation

The other aggregation operator, hash aggregate, is similar to hash join. It does not require (or preserve) sort order, requires memory, and is blocking (that is, it does not produce any results until it has consumed its entire input). Hash aggregate excels at efficiently aggregating very large data sets and, in parallel plans, scales better than stream aggregate.

Here is pseudo-code for the hash aggregate algorithm:

```

for each input row
begin
    calculate hash value on group by column(s)
    check for a matching row in the hash table
    if matching row not found
        insert a new row into the hash table
    else
        update the matching row with the input row
end
output all rows in the hash table

```

While stream aggregate computes just one group at a time, hash aggregate computes all of the groups simultaneously. Like a hash join, a hash aggregate uses a hash table to store these groups. With each new input row, it checks the hash table to see whether the new row belongs to an existing group. If it does, it simply updates the existing group. If it does not, it creates a new group. Since the input data is unsorted, any row can belong to any group. Thus, a hash aggregate cannot output any results until it finishes processing every input row.

Memory and Spilling

As with hash join, the hash aggregate requires memory. Before executing a query with a hash aggregate, SQL Server uses cardinality estimates to estimate how much memory it needs to execute the query. A hash join stores each build row, so the total memory requirement is proportional to the number and size of the build rows. The number of rows that join and the output cardinality of the join have no effect on the memory requirement of the join. A hash aggregate, on the other hand, stores only one row for each group, so the total memory requirement is actually proportional to the number and size of the output groups or rows. If there are fewer unique values of the group by column(s) and fewer groups, a hash aggregate needs less memory. If there are more unique values of the group by column(s) and more groups, a hash aggregate needs more memory.

Like hash join, if a hash aggregate runs out of memory, it must begin spilling rows to a workfile in tempdb. The hash aggregate spills one or more buckets, including any partially aggregated results along with any additional new rows that hash to the spilled buckets. Once the hash aggregate finishes processing all input rows, it outputs the completed in-memory groups and repeats the algorithm by reading back and aggregating one set of spilled buckets at a time. By dividing the spilled rows into multiple sets of buckets, the hash aggregate reduces the size of each set and, thus, reduces the risk that the algorithm will need to repeat many times.

Note that while duplicate rows are a potential problem for hash join, as they lead to skew in the size of the

different hash buckets and make it difficult to divide the work into small uniform portions, duplicates can be quite helpful for hash aggregate since they collapse into a single group.

Examples

The optimizer tends to favor hash aggregation for tables with more rows, fewer groups, no ORDER BY clause or other reason to sort, and no index that produces sorted rows. For example, our original stream aggregate example grouped the Orders table on the ShipAddress and ShipCity columns. This query produces 94 groups from 830 orders. Now consider the following essentially identical query, which groups the Orders table on the ShipCountry column:

```
SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
```

The ShipCountry column has only 21 unique values. Because a hash aggregate requires less memory as the number of groups decreases, whereas a sort requires memory proportional to the number of input rows, this time the optimizer chooses a plan with a hash aggregate:

```
--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
--Hash Match(Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...))
  DEFINE:([Expr1006]=COUNT(*))
  |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Notice that the hash aggregate hashes on the group by column. (The residual predicate from the hash aggregate— which has been edited out of the above text plan to save space—is used to compare rows in the hash table to input rows in case of a hash value collision.) Also observe that with the hash aggregate no sort is needed. However, suppose we explicitly request a sort using an ORDER BY clause in the query:

```
SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]
```

Because of the explicit ORDER BY clause, the plan must include a sort, thus the optimizer chooses a stream aggregate plan:

Code View:

```
--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
  |--Stream Aggregate(GROUP BY:([Orders].[ShipCountry]) DEFINE:([Expr1006]=Count(*)))
    |--Sort(ORDER BY:([Orders].[ShipCountry] ASC))
      |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

If the table gets big enough and the number of groups remains small enough, eventually the optimizer will decide that it is cheaper to use the hash aggregate and sort after the aggregation. For example, our Northwind2 database includes a BigOrders table, which includes the same data from the original Orders table repeated five times. The BigOrders table has 4,150 rows compared to the 830 rows in the original Orders table. However, if

we repeat the above query against the BigOrders table, we still get the same 21 groups:

```
SELECT [ShipCountry], COUNT(*)
FROM [BigOrders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]
```

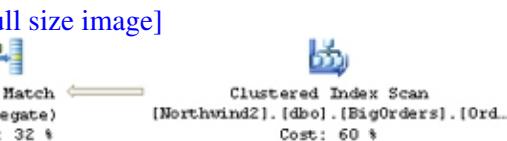
As the following plan shows, the optimizer concludes that it is better to use the hash aggregate and sort 21 rows than to sort 4,150 rows and use a stream aggregate:

Code View:

```
--Sort (ORDER BY:([BigOrders].[ShipCountry] ASC))
--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
--Hash Match (Aggregate, HASH:([BigOrders].[ShipCountry]),
RESIDUAL:(...) DEFINE:([Expr1007]=COUNT(*)))
--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))
```

[Figure 3-13](#) shows the graphical plan for this query. Note the sort is performed as the last operation, as the hash aggregation did not require that the data be sorted earlier.

Figure 3-13. Graphical plan with hash aggregation and sorting



Distinct

Just like stream aggregate, hash aggregate can be used to implement distinct operations. For example, suppose we just want a list of distinct countries to which we've shipped orders:

```
SELECT DISTINCT [ShipCountry] FROM [Orders]
```

Just as the optimizer chose a hash aggregate when we grouped on the ShipCountry column, it also chooses a hash aggregate for this query:

```
--Hash Match (Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...))
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Finally, hash aggregate can be used to implement distinct aggregates, including multiple distincts. The basic idea is the same as for stream aggregate. SQL Server computes each aggregate separately and then joins the results together. For instance, suppose that, for each country to which we have shipped orders, we wish to find

both the number of employees that have taken orders and the number of customers that have placed orders that were shipped to that country. For the optimizer to choose a plan with a hash aggregate, we need to run this query against a much larger table than we've used for our other examples. Our Northwind2 database includes a HugeOrders table, which includes the same data from the original Orders table repeated 25 times.

Code View:

```
SELECT [ShipCountry], COUNT(DISTINCT [EmployeeId]), COUNT(DISTINCT [CustomerId])
FROM [HugeOrders]
GROUP BY [ShipCountry]
```

The following query plan has several interesting features:

Code View:

```
|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].[ShipCountry]))
|--Hash Match(Inner Join, HASH:([HugeOrders].[ShipCountry])=
  ([HugeOrders].[ShipCountry]),RESIDUAL:(...))
|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].[ShipCountry]))
|  |--Compute Scalar(DEFINE:([Expr1005]=CONVERT_IMPLICIT(int,[Expr1012],0)))
|    |--Hash Match(Aggregate, HASH:([HugeOrders].[ShipCountry]),RESIDUAL:(...)
|      DEFINE:([Expr1012]=COUNT([HugeOrders].[CustomerID])))
|      |--Hash Match(Aggregate,HASH:([HugeOrders].[ShipCountry],
  [HugeOrders].[CustomerID]),RESIDUAL:(...))
|        |--Clustered Index Scan (OBJECT:([HugeOrders].[OrderID]))
|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=
  [HugeOrders].[ShipCountry]))
|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1013],0)))
  |--Stream Aggregate (GROUP BY:([HugeOrders].[ShipCountry])
    DEFINE:([Expr1013]=COUNT([HugeOrders].[EmployeeID])))
    |--Sort (ORDER BY:([HugeOrders].[ShipCountry]))
      |--Hash Match(Aggregate,HASH:([HugeOrders].[ShipCountry],
  [HugeOrders].[EmployeeID]),RESIDUAL:(...))
      |--Clustered Index Scan (OBJECT:([HugeOrders].[OrderID]))
```

First, this plan shows that SQL Server can mix hash aggregates and stream aggregates in a single plan. In fact, this plan uses both a hash aggregate and a stream aggregate in the computation of COUNT(DISTINCT [EmployeeId]). The hash aggregate eliminates duplicate values from the EmployeeId column, while the sort and stream aggregate compute the counts. Although SQL Server could use a sort distinct and eliminate the hash aggregate, the sort would take more memory.

Second, SQL Server uses a pair of hash aggregates to compute COUNT(DISTINCT [CustomerId]). This portion of the plan works exactly like the earlier distinct example that used two stream aggregates. The bottommost hash aggregate eliminates duplicate values from the CustomerId column, whereas the topmost computes the counts.

Third, because the hash aggregate does not return rows in any particular order, SQL Server cannot use a merge join without introducing another sort. Instead, the optimizer chooses a hash join for this plan. The graphical plan is shown in [Figure 3-14](#).

Figure 3-14. Graphical plan showing hash aggregation and a hash join to compute multiple distinct aggregates



Unions

There are two types of union queries: UNION ALL and UNION. A UNION ALL query simply combines the results from two or more different queries and returns the results. For example, here is a simple UNION ALL query that returns a list of all employees and customers:

```
SELECT [FirstName] + N' ' + [LastName], [City], [Country] FROM [Employees]
UNION ALL
SELECT [ContactName], [City], [Country] FROM [Customers]
```

The plan for this query uses the concatenation operator:

```
|--Concatenation
|--Compute Scalar(DEFINE:([Expr1003]=([Employees].[FirstName] + N' ') +
[Employees].[LastName]))
|   |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
|--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))
```

The concatenation operator simply executes each of its inputs— it may have more than two—one at a time and returns the results from each input. If we have any employees who also happen to be customers, these individuals will be returned twice by this query.

Now let's consider the following similar query, which outputs a list of all cities and countries in which we have employees and/or customers:

```
SELECT [City], [Country] FROM [Employees]
UNION
SELECT [City], [Country] FROM [Customers]
```

Because this query uses UNION rather than UNION ALL, the query plan must eliminate duplicates. The optimizer uses the same concatenation operator as in the prior plan, but adds a sort distinct to eliminate duplicates:

```
|--Sort(DISTINCT ORDER BY:([Union1006] ASC, [Union1007] ASC))
|--Concatenation
|   |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
|   |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))
```

Another essentially identical alternative plan is to replace the sort distinct with a hash aggregate. A sort distinct requires memory proportional to the number of input rows before the duplicates are eliminated, while a hash aggregate requires memory proportional to the number of output rows after the duplicates are eliminated. Thus, a hash aggregate requires less memory than the sort distinct when there are many duplicates, and the optimizer is more likely to choose a hash aggregate when it expects many duplicates. For example, consider the following query, which combines data from the Orders and BigOrders tables to generate a list of all countries to which we have shipped orders:

```
SELECT [ShipCountry] FROM [Orders]
UNION
SELECT [ShipCountry] FROM [BigOrders]
```

Here is the query plan, which shows the hash distinct:

```
--Hash Match(Aggregate, HASH:([Union1007]), RESIDUAL:(...))
  |--Concatenation
    |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
    |--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))
```

Next suppose that we wish to find a list of all employees and customers sorted by name. We know if we can create appropriate indexes on the Employees and Customers tables that we can get a sorted list of employees or customers. Let's see what happens if we create both indexes and run a UNION ALL query with an ORDER BY clause. For this example, we'll need to create a new table with a subset of the columns from the Employees table and we'll need to create two new indexes. Here is the script to create the new table and indexes along with the test query:

Code View:

```
SELECT [EmployeeId], [FirstName] + N' ' + [LastName] AS [ContactName],
       [City], [Country]
INTO [NewEmployees]
FROM [Employees]
ALTER TABLE [NewEmployees] ADD CONSTRAINT [PK_NewEmployees] PRIMARY KEY ([EmployeeId])
CREATE INDEX [ContactName] ON [NewEmployees]([ContactName])
CREATE INDEX [ContactName] ON [Customers]([ContactName])

SELECT [ContactName] FROM [NewEmployees]
UNION ALL
SELECT [ContactName] FROM [Customers]
ORDER BY [ContactName]
```

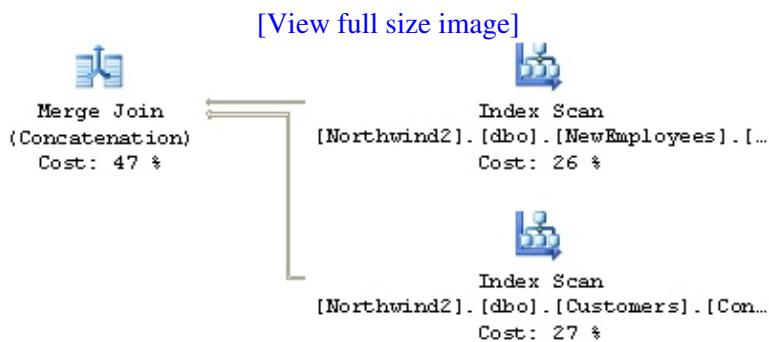
Here is the query plan:

```
--Merge Join(Concatenation)
  |--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
  |--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)
```

Notice that, instead of a concatenation operator, we get a merge join (concatenation) operator. The merge join (concatenation) or merge concatenation operator is not really a join at all. It is implemented by the same iterator as the merge join, but it actually performs a UNION ALL (just like a regular concatenation operator).

while preserving the order of the input rows. Like a merge join, a merge concatenation requires that input data be sorted on the merge key (in this case the ContactName column from the two input tables). By using the order-preserving merge concatenation operator instead of the non-order-preserving concatenation operator, the optimizer avoids the need to add an explicit sort to the plan. The graphical plan is shown in [Figure 3-15](#).

Figure 3-15. Graphical plan showing a merge concatenation



The merge join operator is capable of implementing both UNION ALL, as we have just seen, as well as UNION. For example, let's repeat the previous query as a UNION without the ORDER BY. In other words, we want to eliminate duplicates, but we are not interested in whether the results are sorted.

```

SELECT [ContactName] FROM [NewEmployees]
UNION
SELECT [ContactName] FROM [Customers]

```

The new plan still uses a merge join operator:

Code View:

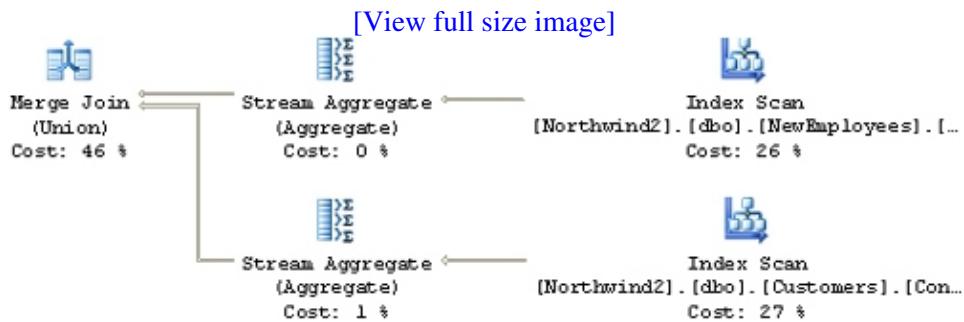
```

|--Merge Join(Union)
|  |--Stream Aggregate(GROUP BY:([NewEmployees].[ContactName]))
|  |  |--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
|  |--Stream Aggregate(GROUP BY:([Customers].[ContactName]))
|  |  |--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)

```

This time the plan has a merge join (union) or merge union operator. The merge union eliminates duplicate rows that appear in both of its inputs; it does not eliminate duplicate rows from either individual input. That is, if there is a name that appears in both the NewEmployees and the Customers tables, the merge union will eliminate that duplicate name. However, if there is a name that appears twice in the NewEmployees table or twice in the Customers table, the merge union by itself will not eliminate it. Thus, we see that the optimizer has also added stream aggregates above each index scan to eliminate any duplicates from the individual tables. As we discussed earlier, aggregation operators can be used to implement distinct operations. You can see the graphical plan in [Figure 3-16](#).

Figure 3-16. Graphical plan showing a merge union operation



Before continuing, let's drop the extra table and indexes that we created just for this example:

```

DROP TABLE [NewEmployees]
DROP INDEX [Customers].[ContactName]
  
```

There is one additional operator that SQL Server can use to perform a UNION. This operator is the hash union, and it is similar to a hash aggregate with two inputs. A hash union builds a hash table on its first input and eliminates duplicates from it just like a hash aggregate. It then reads its second input and, for each row, probes its hash table to see whether the row is a duplicate of a row from the first input. If the row is not a duplicate, the hash union returns it. Note that the hash union does not insert rows from the second input into the hash table. Thus, it does not eliminate duplicates that appear only in its second input. To use a hash union, the optimizer either must explicitly eliminate duplicates from the second input or must know that there are no duplicates in the second input.

Hash unions are rare. To see an example of a hash union, we need to create a large table with big rows but many duplicates. The following script creates two tables. The first table, BigTable, has 100,000 rows, and each row includes a char(1000) column, but all of the rows have the same value for the Dups column. The second table, SmallTable, has a uniqueness constraint to guarantee that there are no duplicates.

Code View:

```

CREATE TABLE [BigTable] ([PK] int PRIMARY KEY, [Dups] int, [Pad] char(1000))
CREATE TABLE [SmallTable] ([PK] int PRIMARY KEY, [NoDups] int UNIQUE, [Pad] char(1000))

SET NOCOUNT ON
DECLARE @i int
SET @i = 0
BEGIN TRAN
WHILE @i < 100000
BEGIN
    INSERT [BigTable] VALUES (@i, 0, NULL)
    SET @i = @i + 1
    IF @i % 1000 = 0
    BEGIN
        COMMIT TRAN
        BEGIN TRAN
    END
END
COMMIT TRAN

SELECT [Dups], [Pad] FROM [BigTable]
UNION
SELECT [NoDups], [Pad] FROM [SmallTable]
  
```

The optimizer chooses a hash union for this query. The hash union is a good choice for eliminating the many duplicates from the BigTable table. Moreover, because of the uniqueness constraint on the NoDups column of the SmallTable table, there is no need to eliminate duplicates from this input to use it with a hash union.

Note

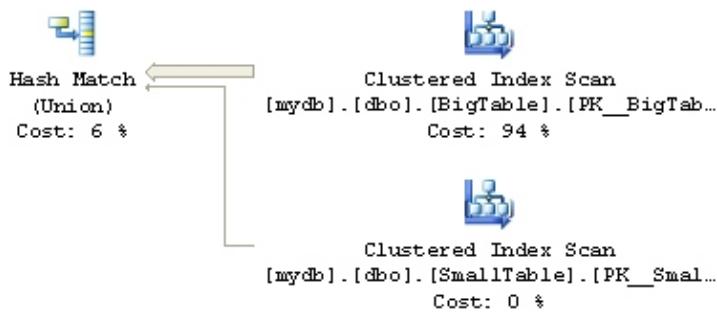


On a machine with multiple processors or multiple cores, the optimizer may choose a substantially different parallel plan that does not include a hash union. If this happens, append an OPTION (MAXDOP 1) hint to the query and you should get the intended plan. The OPTION (MAXDOP 1) query hint forces SQL Server to choose a serial plan for the query. We will discuss parallelism later in this chapter and hints including this one in [Chapters 4](#) and [5](#).

```
|--Hash Match (Union)
|--Clustered Index Scan (OBJECT: ([BigTable].[PK_BigTable]))
|--Clustered Index Scan (OBJECT: ([SmallTable].[PK_SmallTable]))
```

The graphical plan showing the hash union is shown in [Figure 3-17](#).

Figure 3-17. Graphical plan showing a hash union operation



Advanced Index Operations

Earlier in this chapter, we talked about index scans and seeks. You may have noticed that all of the index seek examples we've considered so far have involved simple predicates of the form <column> <comparison operator> <expression>. For instance, our very first example was [OrderDate] = '1998-02-26'. Let's take a look now at how SQL Server uses indexes to execute queries with AND'ed and OR'ed predicates. (AND'ed and OR'ed predicates are often referred to as "conjunctions" and "disjunctions," respectively.) Specifically, we'll look at dynamic index seeks, index unions, and index intersections.

Dynamic Index Seeks

Consider the following query with a simple IN list predicate:

```
SELECT [OrderId]
FROM [Orders]
```

```
WHERE [ShipPostalCode] IN (N'05022', N'99362')
```

We have an index on the ShipPostalCode column, and because this index covers the OrderId column, this query results in a simple index seek:

Code View:

```
--Index Seek(OBJECT:([Orders].[ShipPostalCode]), SEEK:([Orders].[ShipPostalCode]=N'05022'  
OR [Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)
```

Note that the IN list is logically identical to the OR'ed predicate in the index seek. SQL Server executes an index seek with OR'ed predicates by performing two separate index seek operations. First, the server executes an index seek with the predicate [ShipPostalCode] = N'05022' and then it executes a second index seek with the predicate [ShipPostalCode] = N'99362'.

Now consider the following identical query, which uses variables in place of constants:

```
DECLARE @SPC1 nvarchar(20), @SPC2 nvarchar(20)  
SELECT @SPC1 = N'05022', @SPC2 = N'99362'  
SELECT [OrderId]  
FROM [Orders]  
WHERE [ShipPostalCode] IN (@SPC1, @SPC2)
```

We might expect this query to result in a simple index seek as well. However, we instead get the following more complex plan:

Code View:

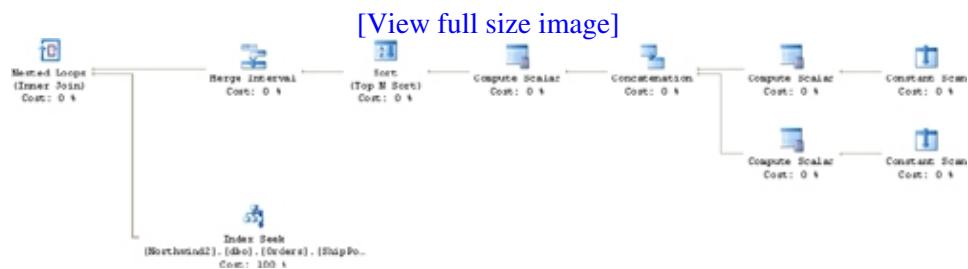
```
--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1009], [Expr1010], [Expr1011]))  
|--Merge Interval  
|   |--Sort(TOP 2, ORDER BY:([Expr1012] DESC, [Expr1013] ASC,  
|                           [Expr1009] ASC, [Expr1014] DESC))  
|   |--Compute Scalar (DEFINE:([Expr1012]=((4)&[Expr1011]) = (4) AND  
|                           NULL = [Expr1009], [Expr1013]=(4)&[Expr1011], [Expr1014]=(16)&[Expr1011]))  
|   |   |--Concatenation  
|   |       |--Compute Scalar(DEFINE:([@SPC1]=[@SPC2], [@SPC2]=[@SPC2],  
|                           [Expr1003]=(62)))  
|   |           |--Constant Scan  
|   |               |--Compute Scalar(DEFINE:([@SPC1]=[@SPC1],  
|                               [@SPC1]=[@SPC1], [Expr1006]=(62)))  
|   |           |--Constant Scan  
|--Index Seek(OBJECT:([Orders].[ShipPostalCode]), SEEK:([Orders].[ShipPostalCode] >  
[Expr1009] AND [Orders].[ShipPostalCode] < [Expr1010]) ORDERED FORWARD)
```

We do not get the index seek that we expect because the optimizer does not know the values of the variables at compile time and, therefore, cannot be sure whether they will have different values or the same value at runtime. If the variables have different values, the original simple index seek plan is valid. However, if the variables have the same value, the original plan is not valid. It would seek to the same index key twice and, thus, return each row twice. Obviously, this result would be incorrect as the query should only return each row

once.

The more complex plan works by eliminating duplicates from the IN list at execution time. The two constant scans and the concatenation operator generate a "constant table" with the two IN list values. Then, the plan sorts the parameter values and the merge interval operator eliminates the duplicates (which, because of the sort, will be adjacent to one another). Finally, the nested loops join executes the index seek once for each unique value. You can see the graphical plan for this query in [Figure 3-18](#).

Figure 3-18. Graphical plan for a query with a dynamic seek for variables in an IN list



You may be wondering why SQL Server needs the merge interval operator. Why couldn't the plan just use a sort distinct to eliminate any duplicate values? To answer this question, let's consider a slightly more complex query:

```
DECLARE @OD1 datetime, @OD2 datetime
SELECT @OD1 = '1998-01-01', @OD2 ='1998-01-04'
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN @OD1 AND DATEADD(day, 6, @OD1)
    OR [OrderDate] BETWEEN @OD2 AND DATEADD(day, 6, @OD2)
```

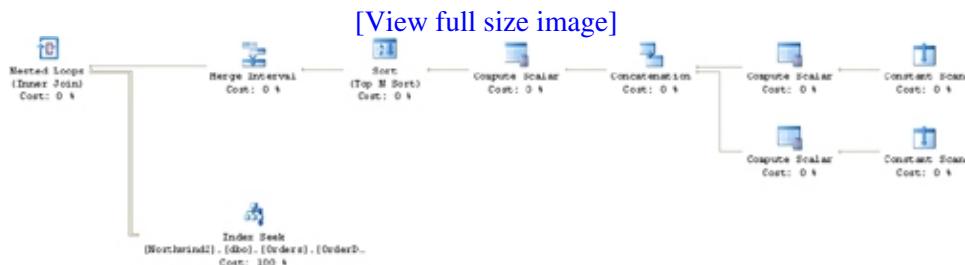
This query returns orders placed within one week of either of two dates. The query plan is nearly identical to the plan for the IN list query:

Code View:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1011],[Expr1012],[Expr1013]))
|  |--Merge Interval
|  |  |--Sort(TOP 2, ORDER BY:([Expr1014] DESC, [Expr1015] ASC,
|  |  |  [Expr1011] ASC, [Expr1016] DESC))
|  |  |--Compute Scalar(DEFINE:([Expr1014]=(4)&[Expr1013]) = (4)
|  |  |  AND NULL = [Expr1011], [Expr1015]=(4)&[Expr1013],
|  |  |  [Expr1016]=(16)&[Expr1013]))
|  |  |--Concatenation
|  |  |  |--Compute Scalar(DEFINE:([@OD1]=[@OD1],
|  |  |  |  [ConstExpr1003]=dateadd(day,(6),[@OD1]),
|  |  |  |  [Expr1007]=(22)|(42)))
|  |  |  |  |--Constant Scan
|  |  |  |--Compute Scalar(DEFINE:([@OD2]=[@OD2],
|  |  |  |  [ConstExpr1004]=dateadd(day,(6),[@OD2]),[Expr1010]=(22)|(42)))
|  |  |  |  |--Constant Scan
|  |  |--Index Seek(OBJECT:([Orders].[OrderDate]),SEEK:([Orders].[OrderDate] > [Expr1011] AND
|  |  |  |  [Orders].[OrderDate] < [Expr1012])) ORDERED FORWARD)
```

Once again, the plan includes a pair of constant scans and a concatenation operator. However, this time instead of returning discrete values from an IN list, the constant scans return ranges. Unlike the prior example, it is no longer sufficient simply to eliminate duplicates. Now the plan needs to handle ranges that are not duplicates but do overlap. The sort ensures that ranges that may overlap are adjacent to one another and the merge interval operator collapses overlapping ranges. In our example, the two date ranges (1998-01-01 to 1998-01-07 and 1998-01-04 to 1998-01-10) do in fact overlap, and the merge interval collapses them into a single range (1998-01-01 to 1998-01-10). The graphical plan for this query is shown in [Figure 3-19](#).

Figure 3-19. Graphical plan for a query with a dynamic seek for variables in a BETWEEN clause



We refer to these plans as dynamic index seeks since the range(s) that SQL Server actually fetches are not statically known at compile time and are determined dynamically during execution. Dynamic index seeks and the merge interval operator can be used for both OR'ed and AND'ed predicates, though they are most common for OR'ed predicates. AND'ed predicates can generally be handled by using one of the predicates, preferably the most selective predicate, as the seek predicate and then applying all remaining predicates as residuals. Recall that with OR'ed predicates we want the union of the set of rows that matches any of the predicates, whereas with AND'ed predicates we want the intersection of the set of rows that matches all of the predicates.

Index Unions

We've just seen how SQL Server can use an index seek even if we have an OR'ed predicate. Next, let's consider a slightly different query that OR's predicates on two different columns:

```

SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
    OR [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'
    
```

This query returns orders that were either placed or shipped during the first week of 1998. We have indexes on both the OrderDate and the ShippedDate columns, but each of these indexes can only be used to satisfy one of the two predicates. If SQL Server uses the index on the OrderDate column to find orders placed during the first week of 1998, it may miss orders that were not placed but did ship during this week. Similarly, if it uses the index on the ShippedDate column to find orders shipped during the first week of 1998, it may miss orders that were placed but did not ship during this week.

It turns out that SQL Server has two strategies that it can use to execute a query such as this one. One option is to use a clustered index scan (or table scan) and apply the entire predicate to all rows in the table. This strategy is reasonable if the predicates are not very selective and if the query will end up returning many rows. However, if the predicates are reasonably selective and if the table is large, the clustered index scan strategy is not very efficient. The other option is to use both indexes. This plan looks as follows:

```

|--Sort (DISTINCT ORDER BY:([Orders].[OrderID] ASC))
| |--Concatenation
|   |--Index Seek (OBJECT:([Orders].[OrderDate]),
|     SEEK:([Orders].[OrderDate] >= '1998-01-01' AND
|       [Orders].[OrderDate] <= '1998-01-07')
|       ORDERED FORWARD)
|   |--Index Seek (OBJECT:([Orders].[ShippedDate]),
|     SEEK:([Orders].[ShippedDate] >= '1998-01-01' AND
|       [Orders].[ShippedDate] <= '1998-01-07')
|       ORDERED FORWARD)

```

The optimizer effectively rewrote the query as a union:

```

SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
UNION
SELECT [OrderId]
FROM [Orders]
WHERE [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'

```

By using both indexes, this plan gets the benefit of the index seek—namely that it only retrieves rows that satisfy the query predicates—yet avoids missing any rows, as might happen if it used only one of the two indexes. However, by using both indexes, the plan may generate duplicates if, as is likely in this example, any of orders were both placed and shipped during the first week of 1998. To ensure that the query plan does not return any rows twice, the optimizer adds a sort distinct. We refer to this type of plan as an index union.

Note that the manual rewrite is only valid because the OrderId column is unique, which ensures that the UNION does not eliminate more rows than it should. However, even if we do not have a unique key and cannot manually rewrite the query, the optimizer always has an internal unique "relational key" for each row (the same key that it uses to perform bookmark lookups) and, thus, can always perform this transformation.

Next, let's consider the following nearly identical query:

```

SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-01-01'
    OR [ShippedDate] = '1998-01-01'

```

The only difference between this query and the prior one is that this query has equality predicates on both columns. Specifically, we are searching for orders that were either placed or shipped on the first day of 1998. Yet, this query yields the following plan:

```

|--Stream Aggregate(GROUP BY:([Orders].[OrderID]))
| |--Merge Join(Concatenation)
|   |--Index Seek (OBJECT:([Orders].[OrderDate]),
|     SEEK:([Orders].[OrderDate]='1998-01-01') ORDERED FORWARD)
|   |--Index Seek (OBJECT:([Orders].[ShippedDate]),
|     SEEK:([Orders].[ShippedDate]='1998-01-01') ORDERED FORWARD)

```

Instead of the concatenation and sort distinct operators, we now have a merge concatenation and a stream

aggregate. Because we have equality predicates on the leading column of each index, the index seeks return rows sorted on the second column (the OrderId column) of each index. Since index seeks return sorted rows, this query is suitable for a merge concatenation. Moreover, since the merge concatenation returns rows sorted on the merge key (the OrderId column), the optimizer can use a stream aggregate instead of a sort to eliminate duplicates. This plan is generally a better choice because the sort distinct uses memory and could spill data to disk if it runs out of memory, while the merge concatenation and stream aggregate do not use memory.

Note that SQL Server did not use the merge concatenation in the prior example because the predicates were inequalities. The inequalities mean that the index seeks in that example returned rows sorted by the OrderDate and the ShippedDate columns. Because the rows were not sorted by the OrderId column, the optimizer could not use the merge concatenation without explicit sorts.

Although the above examples only involve two predicates and two indexes, SQL Server can use index union with any number of indexes, just as we can write a UNION query with any number of inputs.

A union only returns the columns that are common to all of its inputs. In each of the above index union examples (whether based on concatenation and sort distinct, merge union, or hash union), the only column that the indexes have in common is the clustering key OrderId. Thus, the union can only return the OrderId column. If we ask for other columns, the plan must perform a bookmark lookup. This is true even if one of the indexes in the union covers the extra columns. For example, consider the following query, which is identical to the above query but selects the OrderDate and ShippedDate columns in addition to the OrderId column:

```
SELECT [OrderId]
FROM [BigOrders]
WHERE [OrderDate] = '1998-01-01'
    OR [ShippedDate] = '1998-01-01'
```

Note



Note that we need to use the BigOrders table for this example as the optimizer tends to favor simply scanning the entire table instead of performing a bookmark lookup for smaller tables.

Here is the query plan showing the bookmark lookup:

Code View:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([Uniq1002],[BigOrders].[OrderID]))
--Stream Aggregate(GROUP BY:([BigOrders].[OrderID], [Uniq1002]))
|   |--Merge Join(Concatenation)
|       |--Index Seek(OBJECT:([BigOrders].[OrderDate]),
|                       SEEK:([BigOrders].[OrderDate]='1998-01-01')
|                       ORDERED FORWARD)
|       |--Index Seek(OBJECT:([BigOrders].[ShippedDate]),
|                       SEEK:([BigOrders].[ShippedDate]='1998-01-01')
|                       ORDERED FORWARD)
|--Clustered Index Seek(OBJECT:([BigOrders].[OrderID]),
|                       SEEK:([BigOrders].[OrderID]=[BigOrders].[OrderID] AND [Uniq1002]=[Uniq1002])
|                       LOOKUP ORDERED FORWARD)
```

Index Intersections

We've just seen how SQL Server can convert OR'ed predicates into a union query and use multiple indexes to execute this query. SQL Server can also use multiple indexes to execute queries with AND'ed predicates. For example, consider the following query:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26'
    AND [ShippedDate] = '1998-03-04'
```

This query is looking for orders that were placed on February 26, 1998, and were shipped on March 4, 1998. We have indexes on both the OrderDate and the ShippedDate columns, and we can use both indexes. Here is the query plan:

Code View:

```
|--Merge Join(Inner Join, MERGE:([Orders].[OrderID])=([Orders].[OrderID]), RESIDUAL:(...))
|--Index Seek(OBJECT:([Orders].[ShippedDate]),
  SEEK:([Orders].[ShippedDate]='1998-03-04')
  ORDERED FORWARD)
|--Index Seek(OBJECT:([Orders].[OrderDate]),
  SEEK:([Orders].[OrderDate]='1998-02-26')
  ORDERED FORWARD)
```

This plan is very similar to the index union plan with the merge union operator, except that this time we have an actual join. Note that the merge join implements an inner join logical operation. It is really a join this time; it's not a union. The optimizer has effectively rewritten this query as a join (although the explicit rewrite does not get the same plan):

```
SELECT O1.[OrderId]
FROM [Orders] O1 JOIN [Orders] O2
  ON O1.[OrderId] = O2.[OrderId]
WHERE O1.[OrderDate] = '1998-02-26'
    AND O2.[ShippedDate] = '1998-03-04'
```

We refer to this query plan as an index intersection. Just as an index union can use different operators depending on the plan, so can index intersection. As in the merge union example, our first index intersection example uses a merge join because, as a result of the equality predicates, the two index seeks return rows sorted on the OrderId column. Now consider the following query which searches for orders that match a range of dates:

```
SELECT [OrderId]
FROM [BigOrders]
WHERE [OrderDate] BETWEEN '1998-02-01' AND '1998-02-04'
    AND [ShippedDate] BETWEEN '1998-02-09' AND '1998-02-12'
```

Note



We again need to use the larger BigOrders table, as the optimizer favors a simple clustered index scan over a hash-join-based index intersection for the smaller table.

The inequality predicates mean the index seeks no longer return rows sorted by the OrderId column, therefore SQL Server cannot use a merge join without first sorting the rows. Instead of sorting, the optimizer chooses a hash join:

Code View:

```
--Hash Match(Inner Join, HASH:([BigOrders].[OrderID], [Uniq1002])=([BigOrders].[OrderID], [Uniq1002]), RESIDUAL:(...))
--Index Seek(OBJECT:([BigOrders].[OrderDate]),
SEEK:([BigOrders].[OrderDate] >= '1998-02-01' AND [BigOrders].[OrderDate] <= '1998-02-04')
ORDERED FORWARD)
--Index Seek(OBJECT:([BigOrders].[ShippedDate]),
SEEK:([BigOrders].[ShippedDate] >= '1998-02-09' AND [BigOrders].[ShippedDate] <= '1998-02-12')
ORDERED FORWARD)
```

Just like index union, SQL Server can use index intersection plans with more than two tables. Each additional table simply adds one more join to the query plan.

Unlike an index union, which can only deliver those columns that all of the indexes have in common, an index intersection can deliver all of the columns covered by any of the indexes. There is no need for a bookmark lookup. For example, the following query, which selects the OrderDate and ShippedDate columns in addition to the OrderId column, uses the same plan as the similar example that selected just the OrderId column:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26'
AND [ShippedDate] = '1998-03-04'
```

Subqueries

Subqueries are powerful tools that enable us to write far more expressive and far more complex queries. There are many different types of subqueries and many different ways to use subqueries. A complete discussion of subqueries could fill an entire chapter, if not an entire book. In this section, we'll take an introductory look at subqueries.

Subqueries are essentially joins. However, as we'll see, some subqueries generate more complex joins or use some fairly unusual join features.

Before we discuss specific examples, let's look at the different ways that we can classify subqueries. Subqueries can be categorized in three ways:

- Noncorrelated vs. correlated subqueries. A noncorrelated subquery has no dependencies on the outer query, can be evaluated independently of the outer query, and returns the same result for each row of the outer query. A correlated subquery does have a dependency on the outer query. It can only be evaluated in the context of a row from the outer query and may return a different result for each row

of the outer query.

- Scalar vs. multirow subqueries. A scalar subquery returns or is expected to return a single row (that is, a scalar), whereas a multirow subquery may return a set of rows.
- The clause of the outer query in which the subquery appears. Subqueries can be used in nearly any context, including the SELECT list and the FROM, WHERE, ON, and HAVING clauses of the main query.

Noncorrelated Scalar Subqueries

Let's begin our discussion of subqueries by looking at some simple noncorrelated scalar subqueries. The following query returns a list of orders where the freight charge exceeds the average freight charge for all orders:

```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
(
    SELECT AVG(O2.[Freight])
    FROM [Orders] O2
)
```

Notice that we could extract the calculation of the average freight charge and execute it as a completely independent query. Thus, this subquery is noncorrelated. Also, notice that this subquery uses a scalar aggregate and, thus, returns exactly one row. Thus, this subquery is also a scalar subquery. Let's examine the query plan:

```
|--Nested Loops(Inner Join, WHERE:([O1].[Freight]>[Expr1004]))
|--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1011]=(0)
   THEN NULL
   ELSE
      [Expr1012]/CONVERT_IMPLICIT(money,[Expr1011],0) END))
|   |--Stream Aggregate(DEFINE:([Expr1011]=COUNT_BIG([O2].[Freight]),
   [Expr1012]=SUM([O2].[Freight])))
|   |--Clustered Index Scan
      (OBJECT:([Orders].[PK_Orders] AS [O2]))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
```

As you might expect, SQL Server executes this query by first calculating the average freight on the outer side of the nested loops join. The calculation requires a scan of the Orders table (alias [O2]). Since this calculation yields precisely one row, SQL Server then executes the scan on the Orders table (alias [O1]) on the inner side of the join exactly once. The average freight result calculated by the subquery (and stored in [Expr1004]) is used to filter the rows from the second scan.

Now, let's look at another noncorrelated scalar subquery. This time we want to find those orders placed by a specific customer that we've selected by name:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] =
(
    SELECT C.[CustomerId]
    FROM [Customers] C
    WHERE C.[ContactName] = N'Maria Anders'
)
```

Notice that this time the subquery does not have a scalar aggregate to guarantee that it returns exactly one row. Moreover, there is no unique index on the ContactName column, so it is certainly possible that this subquery could actually return multiple rows. However, because the subquery is used in the context of an equality predicate, it is a scalar subquery and must return a single row. If we have two customers with the name "Maria Anders" (which we do not), this query must fail. SQL Server ensures that the subquery returns at most one row by counting the rows with a stream aggregate and then adding an assert operator to the plan:

Code View:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1006]))
| --Assert (WHERE:(CASE WHEN [Expr1005]>(1) THEN (0) ELSE NULL END))
|   |--Stream Aggregate(DEFINE:([Expr1005]=Count(*),
|     [Expr1006]=ANY([C].[CustomerID])))
|       |--Clustered Index Scan (OBJECT:([Customers].[PK_Customers] AS [C]),
|         WHERE:([C].[ContactName]=N'Maria Anders'))
--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
|   SEEK:([O].[CustomerID]=[Expr1006])
|   ORDERED FORWARD)
```

If the assert operator finds that the subquery returned more than one row [that is, if $[Expr1005]>(1)$ is true], it raises the following error:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the
subquery follows =, !=, <, <= , >, >= or when the subquery
is used as an expression.
```

Note that SQL Server uses the assert operator to check many other conditions such as constraints (check, referential integrity, etc.), the maximum recursion level for common table expressions (CTEs), warnings for duplicate key insertions to indexes built with the IGNORE_DUP_KEY option, and more.

The ANY aggregate is a special internal-only aggregate that, as its name suggests, returns any row. Since this plan raises an error if the scan of the Customers table returns more than one row, the ANY aggregate has no real effect. The plan could as easily use the MIN or MAX aggregates and get the same result. However, some aggregate is necessary since the stream aggregate expects each output column either to be aggregated or in the GROUP BY clause (which is empty in this case). This is the same reason that the following query does not compile:

```
SELECT COUNT(*), C.[CustomerId]
FROM [Customers] C
WHERE C.[ContactName] = N'Maria Anders'
```

If you try to execute this query, you get the following error:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Customers.CustomerID' is invalid in the select list because
it is not contained in either an aggregate function or the
GROUP BY clause.
```

The assert operator is not expensive per se and is relatively harmless in this simple example, but it does limit the set of transformations available to the optimizer, which may result in an inferior plan in some cases. Often, creating a unique index or rewriting the query eliminates the assert operator and improves the query plan. For example, as long as we are not concerned that there might be two customers with the same name, the above query can be written as a simple join:

```
SELECT O.[OrderId]
FROM [Orders] O JOIN [Customers] C
  ON O.[CustomerId] = C.[CustomerId]
WHERE C.[ContactName] = N'Maria Anders'
```

This query produces a much simpler join plan:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
 |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]),
    WHERE:([C].[ContactName]=N'Maria Anders'))
 |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
    SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
```

While writing this query as a simple join is the best option, let's see what happens if we create a unique index on the ContactName column:

```
CREATE UNIQUE INDEX [ContactName] ON [Customers] ([ContactName])

SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] =
(
  SELECT C.[CustomerId]
  FROM [Customers] C
  WHERE C.[ContactName] = N'Maria Anders'
)
DROP INDEX [Customers].[ContactName]
```

Because of the unique index, the optimizer knows that the subquery can produce only one row, eliminates the now unnecessary stream aggregate and assert operators, and converts the query into a join:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
 |--Index Seek(OBJECT:([Customers].[ContactName] AS [C]),
    SEEK:([C].[ContactName]=N'Maria Anders') ORDERED FORWARD)
 |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
    SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
```

Correlated Scalar Subqueries

Now that we've seen how SQL Server evaluates a simple noncorrelated subquery, let's explore what happens if we have a correlated scalar subquery. The following query is similar to the first subquery we tried, but this time it returns those orders in which the freight charge exceeds the average freight charge for all previously

placed orders:

```
SELECT O1.[OrderId]
FROM [Orders] O1
WHERE O1.[Freight] >
(
    SELECT AVG(O2.[Freight])
    FROM [Orders] O2
    WHERE O2.[OrderDate] < O1.[OrderDate]
)
```

This time SQL Server cannot execute the subquery independently. Because of the correlation on the OrderDate column, the subquery returns a different result for each row from the main query. Recall that with the noncorrelated subquery, SQL Server evaluated the subquery first and then executed the main query. This time SQL Server evaluates the main query first and then evaluates the subquery once for each row from the main query:

```
--Filter(WHERE:([O1].[Freight]>[Expr1004]))
--Nested Loops(Inner Join, OUTER REFERENCES:([O1].[OrderDate]))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
|--Index Spool SEEK:([O1].[OrderDate]=[O1].[OrderDate])
|--Compute Scalar(DEFINE:([Expr1004]=
CASE WHEN [Expr1011]=(0)
THEN NULL
ELSE [Expr1012]
/CONVERT_IMPLICIT(money,[Expr1011],0) END))
|--Stream Aggregate (DEFINE:([Expr1011]=
COUNT_BIG([O2].[Freight]),[Expr1012]=SUM([O2].[Freight])))
|--Index Spool SEEK:([O2].[OrderDate]<[O1].[OrderDate])
|--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [O2]))
```

This plan is not as complicated as it looks. You can see the graphical plan in [Figure 3-20](#). The index spool immediately above the scan of [O2] is an eager index spool or index-on-the-fly spool. It builds a temporary index on the OrderDate column of the Orders table. It is called an eager spool because it "eagerly" loads its entire input set and builds the temporary index as soon as it is opened.

Figure 3-20. Graphical plan showing two index spool operations: one lazy and one eager



The index makes subsequent evaluations of the subquery more efficient since there is a predicate on the OrderDate column. The stream aggregate computes the average freight charge for each execution of the subquery. The index spool above the stream aggregate is a lazy index spool. It merely caches subquery results. If it encounters any OrderDate a second time, it returns the cached result rather than recomputing the subquery. It is called a lazy spool because it "lazily" loads results on demand only. Finally, the filter at the top of the plan compares the freight charge for each order to the subquery result ([Expr1004]) and returns those rows that qualify.

Note



We can determine the types of the spools more easily from the complete SHOWPLAN_ALL output or from the graphical plan. The logical operator for the spool indicates whether each spool is an eager or lazy spool. The spools are there strictly for performance. The optimizer decides whether to include them in the plan based on its cost estimates. If the optimizer does not expect many duplicate values for the OrderDate column or many rows at all from the outer side of the join, it may eliminate the spools. For example, if you try the same query with a selective filter on the main query such as O1.[ShipCity] = N'Berlin', the spools go away.

We have already seen how a nested loops join executes its inner input once for each row from its outer input. In most cases, each execution of the inner input proceeds completely independently of any prior executions. However, spools are special. A spool, such as the lazy index spool in the above example, is designed to optimize the case in which the inner side of the join executes with the same correlated parameter(s) multiple times. Thus, for a spool, it is useful to distinguish between executions with the same correlated parameter(s) or "rewinds" and executions with different correlated parameters or "rebinds." Specifically, a rewind is defined as an execution with the same correlated parameter(s) as the immediately preceding execution, whereas a rebind is defined as an execution with different correlated parameters than the immediately preceding execution.

A rewind results in the spool playing back a cached result, while a rebind results in the spool "binding" new correlated parameter values and loading a new result. A regular (nonindex) lazy spool only caches one result set at a time. Thus, a regular spool truncates and "reloads" its worktable on each rebind. A lazy index spool, such as the spool in the above plan, accumulates results and does not truncate its worktable on a rebind.

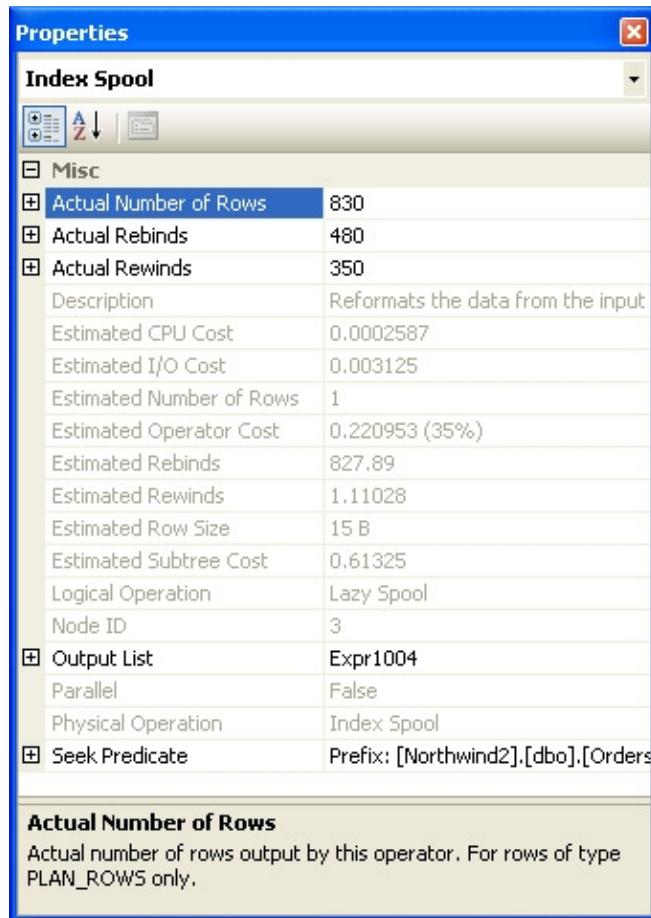
We can see the estimated and actual number of rewinds and rebinds using SET STATISTICS XML ON or using the graphical plan. For example, here are the statistics XML for the lazy index spool. Notice that, as you might expect, the sum of the number of rewinds and rebinds is the same as the total number of executions

Code View:

```
<RelOp NodeId="3" PhysicalOp="Index Spool" LogicalOp="Lazy Spool"...
      >
      <RunTimeInformation>
          <RunTimeCountersPerThread Thread="0" ActualRows="830"
                                         ActualEndOfScans="0"
                                         ActualExecutions="830" />
      </RunTimeInformation>
</RelOp>
```

The ToolTip from the graphical plan, showing the same information as in the above XML fragment, is shown in [Figure 3-21](#).

Figure 3-21. ToolTip showing rewind and rebind information



Note that rewinds and rebinds are counted the same way for index and nonindex spools. As described previously, a reexecution is counted as a rewind only if the correlated parameter(s) remain the same as the immediately prior execution and is counted as a rebind if the correlated parameter(s) change from the prior execution. This is true even for reexecutions, in which the same correlated parameter(s) were encountered in an earlier, though not the immediately prior, execution. However, since lazy index spools like the one in this example retain results for all prior executions and all previously encountered correlated parameter values, the spool may treat some reported rebinds as rewinds. In other words, by failing to account for correlated parameter(s) that were seen prior to the most recent execution, the query plan statistics may overreport the number of rebinds for an index spool.

Next let's look at another example of a correlated scalar subquery. Suppose that we wish to find those orders for which the freight charge exceeds the average freight charge for all orders placed by the same customer:

```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
(
    SELECT AVG(O2.[Freight])
    FROM [Orders] O2
    WHERE O2.[CustomerId] = O1.[CustomerId]
)
```

This query is very similar to the previous one, yet we get a substantially different plan:

Code View:

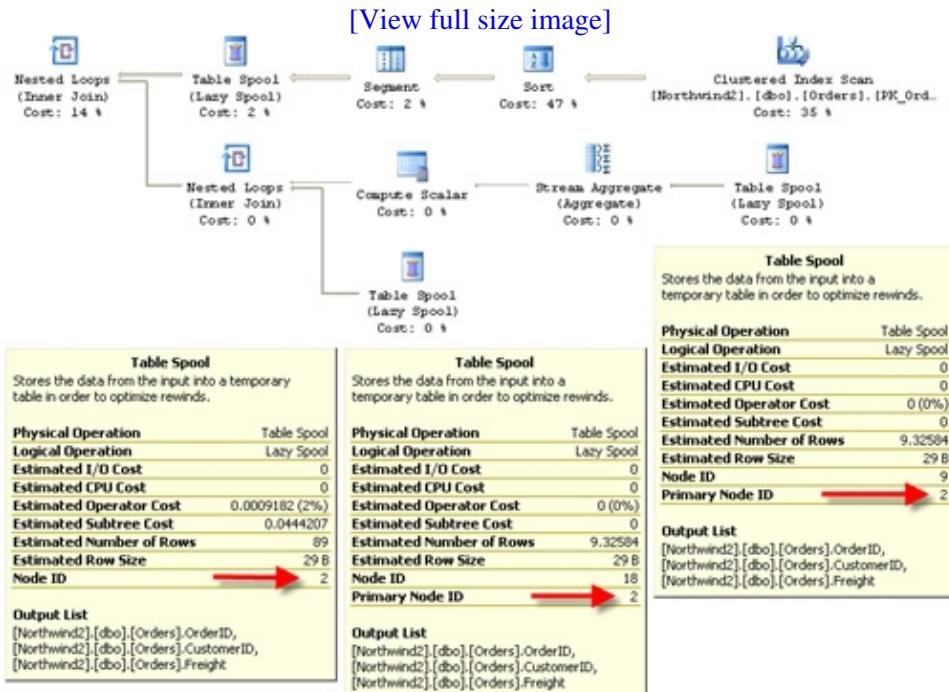
```

|--Nested Loops(Inner Join)
|  |--Table Spool
|  |  |--Segment
|  |  |  |--Sort (ORDER BY:([O1].[CustomerID] ASC))
|  |  |  |--Clustered Index Scan (OBJECT:([Orders].[PK_Orders] AS [O1]),
|  |  |  |  WHERE:([O1].[CustomerID] IS NOT NULL))
|--Nested Loops(Inner Join, WHERE:([O1].[Freight]>[Expr1004]))
|  |--Compute Scalar(DEFINE:([Expr1004]=
|  |  CASE WHEN [Expr1012]=(0)
|  |  |  THEN NULL
|  |  |  ELSE [Expr1013]
|  |  |  /CONVERT_IMPLICIT(money,[Expr1012],0)
|  |  |  END))
|  |--Stream Aggregate(DEFINE:([Expr1012]=
|  |  COUNT_BIG([O1].[Freight]),[Expr1013]=SUM([O1].[Freight])))
|  |  |--Table Spool
|  |  |--Table Spool

```

Again, this plan is not as complicated as it looks and you can see the graphical plan in Figure 3-22. The outer side of the topmost nested loops join sorts the rows of the clustered index scan by the CustomerId column. The segment operator breaks the rows into groups (or segments) with the same value for the CustomerId column. Since the rows are sorted, sets of rows with the same CustomerId value will be consecutive. Next, the table spoolâ a segment spoolâ reads and saves one of these groups of rows that share the same CustomerId value.

Figure 3-22. Graphical plan showing a segment spool with two secondary spools



When the spool finishes loading a group of rows, it returns a single row for the entire group. (Note that a segment spool is the only type of spool that exhibits this behavior of returning only a single row regardless of how many input rows it reads.) At this point, the topmost nested loops join executes its inner input. The two

leaf-level table spoolsâ secondary spoolsâ replay the group of rows that the original segment spool saved. The graphical plan illustrates the relationship between the secondary and primary spools by showing that the Primary Node ID for each of the secondary spools is the same as the Node ID for the primary segment spool. (In this example, the primary spool's Node ID is 2.) The stream aggregate computes the average freight for each group of rows (and, thus, for each CustomerId value). The result of the stream aggregate is a single row. The inner of the two nested loops compares each spooled row (which again consists of rows with the same CustomerId value) against this average and returns those rows that qualify. Finally, the segment spool truncates its worktable and repeats the process beginning with reading the next group of rows with the next CustomerId value.

By using the segment spool, the optimizer creates a plan that needs to scan the Orders table only one time. We refer to a spool such as this one that replays the same set of rows in different places within the plan as a common subexpression spool. Note that not all common subexpression spools are segment spools.

Note



We know that the spool is a segment spool, as it appears immediately above a segment operator in the plan. As in the prior example, we can find more information about the spools and other operators in this plan from SHOWPLAN_ALL, SHOWPLAN_XML, or the graphical plan. SHOWPLAN_XML and the graphical plan provide the richest information including the group by column for the segment operator and the primary spool's Node ID for each secondary spool.

Finally, suppose that we want to compute the order with the maximum freight charge placed by each customer:

```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] =
(
    SELECT MAX(O2.[Freight])
    FROM [Orders] O2
    WHERE O2.[CustomerId] = O1.[CustomerId]
)
```

This query is similar to the previous two queries, yet we once again get a very different and surprisingly simple plan:

```
--Top(TOP EXPRESSION: ((1)))
--Segment
--Sort(ORDER BY: ([O1].[CustomerID] DESC, [O1].[Freight] DESC))
--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [O1]),
 WHERE:([O1].[Freight] IS NOT NULL AND
 [O1].[CustomerID] IS NOT NULL))
```

This plan sorts the Orders table by the CustomerId and Freight columns. As in the previous example, the segment operator breaks the rows into groups or segments with the same value for the CustomerId column. The top operator is a segment top. Unlike a normal top, which returns the top N rows for the entire input set, a segment top returns the top N rows for each group. The top is also a "top with ties." A top with ties returns more than N rows if there are duplicates or ties for the Nth row. In this query plan, since the sort assures that

the rows with the maximum freight charge are ordered first within each group, the top returns the row or rows with the maximum freight charge from each group. This plan is very efficient since it processes the Orders table only once and, unlike the previous plan, does not need a spool.

Note



As with the segment spool, we know that the top is a segment top because it appears immediately above a segment operator. We can also tell that the top is a top with ties by checking SHOWPLAN_ALL, SHOWPLAN_XML, or the graphical plan.

Removing Correlations

The SQL Server query optimizer is able to remove correlations from many subqueries including scalar and multirow queries. If the optimizer is unable to remove correlations from a subquery, it must execute the subquery plan on the inner side of a nested loops join. However, by removing correlations, the optimizer can transform a subquery into a regular join and consider more plan alternatives with different join orders and join types. For example, the following query uses a noncorrelated subquery to return orders placed by customers who live in London:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] IN
(
    SELECT C.[CustomerId]
    FROM [Customers] C
    WHERE C.[City] = N'London'
)
```

We can easily write the same query using a correlated subquery:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE EXISTS
(
    SELECT *
    FROM [Customers] C
    WHERE C.[CustomerId] = O.[CustomerId]
        AND C.[City] = N'London'
)
```

One of these queries includes a noncorrelated subquery whereas the other includes a correlated subquery, however the optimizer generates the same plan for both queries:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
--Index Seek(OBJECT:([Customers].[City] AS [C]),
SEEK:([C].[City]=N'London')
ORDERED FORWARD)
--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
SEEK:([O].[CustomerID]=[C].[CustomerID])
ORDERED FORWARD)
```

In the case of the second query, the optimizer removes the correlation from the subquery so that it can scan the Customers table first (on the outer side of the nested loops join). If the optimizer did not remove the correlation, the plan would need to scan the Orders table first to generate a CustomerId value before it could scan the Customers table.

For a more complex example of subquery decorrelation, consider the following query, which outputs a list of orders along with the freight charge for each order and the average freight charge for all orders by the same customer:

```
SELECT O1.[OrderId], O1.[Freight],
(
    SELECT AVG(O2.[Freight])
    FROM [Orders] O2
    WHERE O2.[CustomerId] = O1.[CustomerId]
) Avg_Freight
FROM [Orders] O1
```

We might expect that a correlated SELECT list subquery, like the one used in this query, must be evaluated exactly once for each row from the main query. However, as the following plan illustrates, the optimizer is able to remove the correlation from this query:

Code View:

```
--Compute Scalar(DEFINE:([Expr1004]=[Expr1004]))
--Hash Match(Right Outer Join,HASH:([O2].[CustomerID])=([O1].[CustomerID]),
RESIDUAL:(...))
--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1013]=(0)
THEN NULL
ELSE [Expr1014]
/CONVERT_IMPLICIT(money,[Expr1013],0)
END))
|   |--Stream Aggregate(GROUP BY:([O2].[CustomerID])
DEFINE:([Expr1013]=COUNT_BIG([O2].[Freight]),
[Expr1014]=SUM([O2].[Freight])))
|   |--Sort(ORDER BY:([O2].[CustomerID] ASC))
|       |--Clustered Index Scan
|           (OBJECT:([Orders].[PK_Orders] AS [O2]))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
```

This plan first computes the average freight charge for all customers and then joins this result with the Orders table. Note how this plan computes the average freight charge for each customer exactly once regardless of the number of orders placed by that customer. Had the optimizer not removed the correlation, the plan would have had to compute the average freight charge for each customer separately for each order placed by that customer. For example, if a customer placed three orders, the plan would have computed the average freight charge for that customer three times. Clearly, the plan with the decorrelated subquery is more efficient. On the other hand, if we have a sufficiently selective predicate on the main query, the correlated plan does become more efficient and the optimizer will select it.

In addition to computing the average freight charges only once per customer, by removing the correlation, the optimizer is free to use any join operator. For example, this plan uses a hash join. The join itself is a right outer join. The outer join guarantees that the plan returns all orders, even those that might have a NULL value for the CustomerId column. An inner join would discard such rows since NULLs never join.

Subqueries in CASE Expressions

Normally, SQL Server evaluates CASE expressions like any other scalar expression, often using a compute scalar operator. In the absence of any subqueries, there is really nothing remarkable about a CASE expression. However, it is possible to use subqueries in the WHEN, THEN, and ELSE clauses of a CASE expression. SQL Server uses some slightly more exotic join functionality, which we have not seen yet, to evaluate CASE expressions with subqueries. To see how these plans work, we'll use the following script to set up an artificial scenario:

Code View:

```

CREATE TABLE [MainTable] ([PK] int PRIMARY KEY, [Col1] int, [Col2] int, [Col3] int)
CREATE TABLE [WhenTable] ([PK] int PRIMARY KEY, [Data] int)
CREATE TABLE [ThenTable] ([PK] int PRIMARY KEY, [Data] int)
CREATE TABLE [ElseTable] ([PK] int PRIMARY KEY, [Data] int)

INSERT [MainTable] VALUES (1, 11, 101, 1001)
INSERT [MainTable] VALUES (2, 12, 102, 1002)
INSERT [WhenTable] VALUES (11, NULL)
INSERT [ThenTable] VALUES (101, 901)
INSERT [ElseTable] VALUES (102, 902)
SELECT M.[PK],
CASE WHEN EXISTS (SELECT * FROM [WhenTable] W WHERE W.[PK] = M.[Col1])
    THEN (SELECT T.[Data] FROM [ThenTable] T WHERE T.[PK] = M.[Col2])
    ELSE (SELECT E.[Data] FROM [ElseTable] E WHERE E.[PK] = M.[Col3])
END AS Case_Expr
FROM [MainTable] M

DROP TABLE [MainTable], [WhenTable], [ThenTable], [ElseTable]

```

Semantically, this query scans table MainTable and for each row checks whether there is a matching row in table WhenTable. If there is a match, it looks up an output value in table ThenTable; otherwise, it looks up an output value in table ElseTable. To show precisely what is happening, the script also adds a few rows of data. MainTable has two rows. One of these rows matches a row in WhenTable, whereas the other does not. Thus, one row results in a lookup from ThenTable while the other row results in a lookup from ElseTable. Finally, ThenTable includes rows that match both MainTable rows, while ElseTable is empty. Although ElseTable is empty, the query still outputs all rows from MainTable, including the row that does not have a match in WhenTable. This row simply outputs NULL for the result of the CASE expression. Here is the output of the query:

PK	Case_Expr
1	901
2	NULL

And here is the query plan:

Code View:

```

Rows Executes
0   0   |--Compute Scalar(DEFINE:([Expr1011]=CASE WHEN [Expr1012]
      THEN [T].[Data] ELSE [E].[Data] END))
1     |--Nested Loops(Left Outer Join, ,
      OUTER REFERENCES:([M].[Col3]))
1       |--Nested Loops(Left Outer Join,
      OUTER REFERENCES:([M].[Col2]))
1         |--Nested Loops(Left Semi Join,

```

```

        OUTER REFERENCES: ([M].[Col1]),
)
1     |   |   |--Clustered Index Scan (OBJECT: ([MainTable].[PK_MainTable]
      AS [M]))
1     |   |   |--Clustered Index Seek (OBJECT: ([WhenTable].[PK_WhenTable]
      AS [W]), SEEK: ([W].[PK]=[M].[Col1]))
1     |   |   |--Clustered Index Seek (OBJECT: ([ThenTable].[PK_ThenTable] AS [T]),
      SEEK: ([T].[PK]=[M].[Col2]))
0     |   |--Clustered Index Seek (OBJECT: ([ElseTable].[PK_ElseTable] AS [E]),
      SEEK: ([E].[PK]=[M].[Col3]))

```

As we might expect, this query plan begins by scanning MainTable. This scan returns two rows. Next, the plan executes the WHEN clause of the CASE expression. The plan implements the EXISTS subquery using a left semi-join with WhenTable. SQL Server frequently uses semi-joins to evaluate EXISTS subqueries since the semi-join merely checks whether a row from one input joins with or matches any row from the other input. However, a normal semi-join (or anti-semi-join) only returns rows for matches (or nonmatches). In this case, the query must return all rows from MainTable, regardless of whether these rows have a matching row in WhenTable. Thus, the semi-join cannot simply discard a row from MainTable just because WhenTable has no matching row.

The solution is a special type of semi-join with a PROBE column. This semi-join returns all rows from MainTable whether or not they match and sets the PROBE column (in this case [Expr1012]) to true or false to indicate whether or not it found a matching row in WhenTable. Since the semi-join does not actually return a row from WhenTable, without the PROBE column there would be no way to determine whether or not the semi-join found a match.

Next, depending on the value of the PROBE column, the query plan needs to look for a matching row in either ThenTable or ElseTable. However, the query plan must look in only one of the two tables. It cannot look in both. The plan uses a special type of nested loops join to ensure that it performs only one of the two lookups. This nested loops join has a special predicate known as a PASSTHRU predicate. The join evaluates the PASSTHRU predicate on each outer row. If the PASSTHRU predicate evaluates to true, the join immediately returns the outer row without executing its inner input. If the PASSTHRU predicate evaluates to false, the join proceeds normally and tries to join the outer row with an inner row.

Note



In this example, the query plan could execute both subqueries (for the THEN and ELSE clauses of the CASE expression) and then discard any unnecessary results. However, besides being inefficient, in some cases executing the extra subqueries could cause the query plan to fail. For example, if one of the scalar subqueries joined on a nonunique column, it could return more than one row, which would result in an error. It would be incorrect for the plan to fail while executing an unnecessary operation.

The plan actually has two nested loops joins with PASSTHRU predicates: one to evaluate the THEN clause subquery and one to evaluate the ELSE clause subquery. The PASSTHRU predicate for the first (bottommost) join tests the condition IsFalseOrNull [Expr1012]. The IsFalseOrNull function simply inverts the Boolean PROBE column. For each MainTable row, if the semi-join finds a matching row, the PROBE column ([Expr1012]) is true, the PASSTHRU predicate evaluates to false, and the join evaluates the index seek on ThenTable. However, if the semi-join does not find a matching row, the PROBE column is false, the PASSTHRU predicate evaluates to true, and the join returns the MainTable row without evaluating the index

seek on ThenTable. The PASSTHRU predicate for the next (topmost) join tests the opposite condition to determine whether to perform the index seek on ElseTable. Thus, the plan executes exactly one of the two index seeks for each MainTable row.

We can see the behavior of the PASSTHRU predicates by observing that while the MainTable scan and each of the joins returns two rows, the plan executes the index seeks on ThenTable and ElseTable only once. A PASSTHRU predicate is the only scenario in which the number of rows on the outer side of a nested loops join does not precisely match the number of executes on the inner side.

Also notice how the query plan uses outer joins since there is no guarantee that the THEN or ELSE subqueries will actually return any rows. In fact, in this example, the index seek on ElseTable is executed for one of the MainTable rows yet returns no rows. The outer join simply returns a NULL and the query still returns the MainTable row. If the query plan had used an inner join, it would have incorrectly discarded the MainTable row.

Tip



The above query plan was generated using SQL Server 2005. If you run this example on SQL Server 2000, you will still get a plan with a PASSTHRU predicate, but it will appear in the plan as a regular WHERE clause predicate. Unfortunately, on SQL Server 2000, there is no easy way to differentiate a regular WHERE clause predicate from a PASSTHRU predicate.

This example demonstrates a CASE expression with a single WHEN clause. SQL Server supports CASE expressions with multiple WHEN clauses and multiple THEN clause subqueries in the same way. The PASSTHRU predicates merely get progressively more complex to ensure that only one of the THEN clauses (or the ELSE clause) is actually executed.

Parallelism

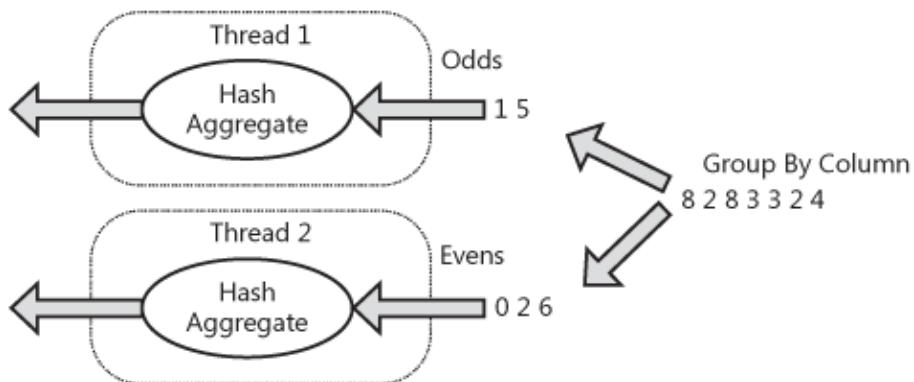
SQL Server has the ability to execute queries using multiple CPUs simultaneously. We refer to this capability as parallel query execution. Parallel query execution can be used to reduce the response time of (that is, speed up) a large query. It can also be used to run a bigger query (one that processes more data) in about the same amount of time as a smaller query (that is, scale up) by increasing the number of CPUs used in processing the query.

While parallelism can be used to reduce the response time of a single query, this speedup comes at a cost: It increases the overhead associated with executing a query. While this overhead is relatively small, it does make parallelism inappropriate for small queries (e.g., for OLTP queries) in which the overhead would dominate the total execution time and the goal is to run the maximum number of concurrent queries and to maximize the overall throughput of the system. SQL Server does generally scale well; however, if we compare the same query running serially (that is, without parallelism and on a single CPU) and in parallel on two CPUs, we will typically find that the parallel execution time is more than half of the serial execution time. Again, this effect is caused by the parallelism overhead.

Parallelism is primarily useful on servers running a relatively small number of concurrent queries. On this type of server, parallelism can enable a small set of queries to keep many CPUs busy. On servers running many concurrent queries (such as an OLTP system), we do not need parallelism to keep the CPUs busy; the mere fact that we have so many queries to execute can keep the CPUs busy. As we've already discussed, running these queries in parallel would just add overhead that would reduce the overall throughput of the system.

SQL Server parallelizes queries by horizontally partitioning the input data into approximately equal-sized sets, assigning one set to each CPU, and then performing the same operation (for instance, aggregate, join, etc.) on each set. For example, suppose that SQL Server decides to use two CPUs to execute a hash aggregate that happens to be grouping on an integer column. The server creates two threads (one for each CPU). Each thread executes the same hash aggregate operator. SQL Server might partition the input data by sending rows in which the GROUP BY column is odd to one thread and rows in which the GROUP BY column is even to the other thread, as illustrated in [Figure 3-23](#). As long as all rows that belong to one group are processed by one hash aggregate operator and one thread, the plan produces the correct result.

Figure 3-23. Executing an aggregate query on two threads



This method of parallel query execution is both simple and scales well. In the above example, both hash aggregate threads execute independently. The two threads do not need to communicate or coordinate their work in any way. To increase the degree of parallelism (DOP), SQL Server can simply add more threads and adjust the partitioning function. In practice, SQL Server uses a hash function to distribute rows for a hash aggregate. The hash function handles any data type, any number of GROUP BY columns, and any number of threads.

Note that this method of parallelism is not the same as "pipeline" parallelism in which multiple unrelated operators run concurrently in different threads. Although SQL Server frequently places different operators in different threads, the primary reason for doing so is to allow repartitioning of the data as it flows from one operator to the next. With pipeline parallelism, the degree of parallelism and the total number of threads would be limited to the number of operators.

The query optimizer decides whether to execute a query in parallel. For the optimizer even to consider a parallel plan, the following criteria must be met:

- SQL Server must be running on a multiprocessor, multicore, or hyperthreaded machine.
- The affinity mask (and affinity mask64 for 64 processor servers) advanced configuration option must allow SQL Server to use at least two processors. The default setting of zero for affinity mask allows SQL Server to use all available processors.
- The max degree of parallelism advanced configuration option must be set to zero (the default) or to more than one. We discuss this option in more detail below.

Like most other decisions, the choice of whether to choose a serial or a parallel plan is cost-based. A complex and expensive query that processes many rows is more likely to result in a parallel plan than a simple query that processes very few rows. Although it is rarely necessary, you can also adjust the cost threshold for parallelism advanced configuration setting to raise or lower the threshold above which the optimizer considers parallel plans.

Degree of Parallelism (DOP)

The DOP is not part of the cached compiled plan and may change with each execution. SQL Server decides the DOP at the start of execution as follows:

1. If the query includes a MAXDOP N query hint, SQL Server sets the maximum DOP for the query to N or to the number of available processors (limited by the affinity mask configuration option) if N is zero.
 2. If the query does not include a MAXDOP N query hint, SQL Server sets the maximum DOP to the setting of the max degree of parallelism advanced configuration option.
- As with the MAXDOP N query hint, if this option is set to zero (the default), SQL Server sets the maximum DOP to the number of available processors.
3. SQL Server calculates the maximum number of concurrent threads that it needs to execute the query plan (as we will see momentarily, this number can exceed the DOP) and compares this result to the number of available threads. If there are not enough available threads, SQL Server reduces the DOP, as necessary. In the extreme case, SQL Server switches the parallel plan back to a serial plan. A serial plan runs with just the single connection thread and, thus, can always execute. Once the DOP is fixed, SQL Server reserves sufficient threads to ensure that the query can execute without running out of threads. This process is similar to how SQL Server acquires a memory grant for memory-consuming queries. The principle difference is that a query may wait to acquire a memory grant but never waits to reserve threads.

The max worker threads advanced configuration option determines the maximum number of threads available to SQL Server for system activities, as well as for parallel query execution. The default setting for this option varies depending on the number of processors. It is set higher for systems with more processors. It is also set higher for 64-bit servers than for 32-bit servers. It is rarely necessary to change the setting of this option.

As noted above, the number of threads used by a query may exceed the DOP. If you check sys.dm_os_tasks while running a parallel query, you may see more threads than the DOP. The number of threads may exceed the DOP because, if SQL Server needs to repartition data between two operators, it places them in different threads. The DOP only determines the number of threads per operator, not the total number of threads per query plan. As with the memory grant computation, when SQL Server computes the maximum number of threads that a query plan may consume, it does take blocking or stop-and-go operators into account. The operators above and below a blocking operator are never executed at the same time and, thus, can share both memory and threads.

In SQL Server 2000 if the DOP was less than the number of CPUs, the extra threads could use the extra CPUs, effectively defeating the MAXDOP settings. In SQL Server 2005, when a query runs with a given DOP, SQL Server also limits the number of schedulers used by that query to the selected DOP. That is, all threads used by the query are assigned to the same set of DOP schedulers, and the query uses only DOP CPUs, regardless of the total number of threads.

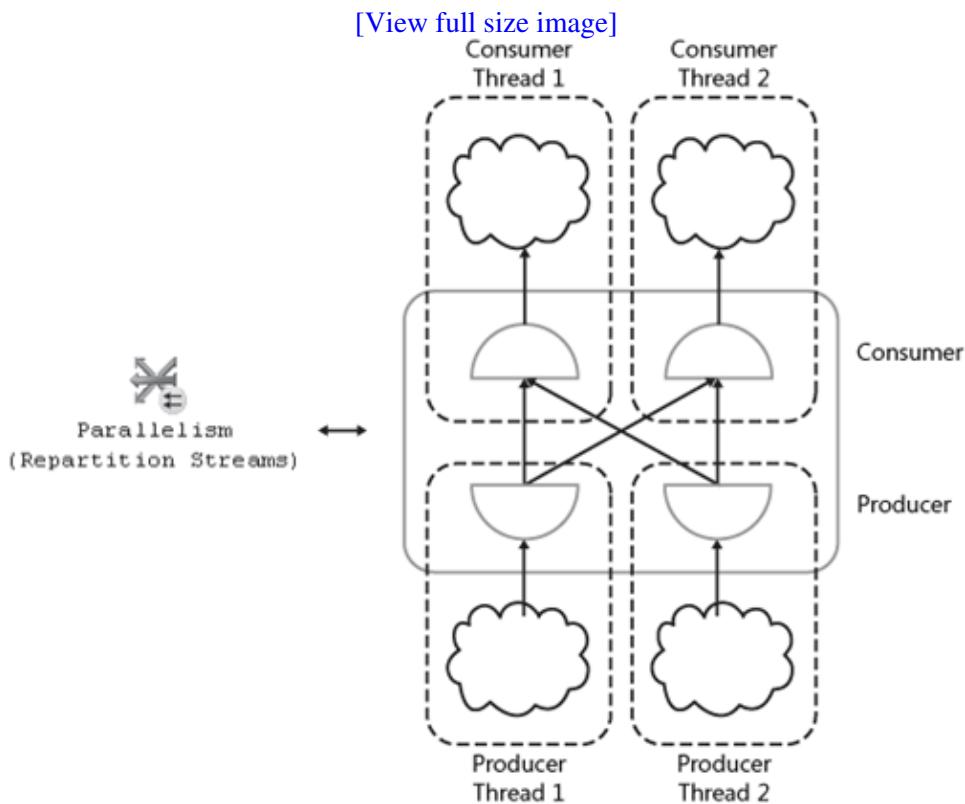
The Parallelism Operator (also known as Exchange)

The actual partitioning and movement of data between threads is handled by the parallelism (or exchange) iterator. Although it is unique in many respects, the parallelism iterator implements the same interfaces as any other iterator. Most of the other iterators do not need to be aware that they are executing in parallel. The optimizer simply places appropriate parallelism iterators in the plan and it runs in parallel.

The exchange iterator is unique in that it is really two iterators: a producer and a consumer. SQL Server places the producer at the root of a query subtree (often called a branch). The producer reads input rows from its subtree, assembles the rows into packets, and routes these packets to the appropriate consumer(s). SQL Server places the consumer at the leaf of the next query subtree. The consumer receives packets from its producer(s),

removes the rows from these packets, and returns the rows to its parent iterator. For example, as Figure 3-24 illustrates, a repartition exchange running at DOP2 consists of two producers and two consumers:

Figure 3-24. A repartition exchange running on two CPUs



Note that while the data flow between most iterators is pull-based (an iterator calls GetRow on its child when it is ready for another row), the data flow in between an exchange producer and consumer is push-based. That is, the producer fills a packet with rows and "pushes" it to the consumer. This model allows the producer and consumer threads to execute independently. SQL Server does have flow control to prevent a fast producer from flooding a slow consumer with excessive packets.

Exchanges can be classified in three different ways:

- by the number of producer and consumer threads,
- by the partitioning function used by the exchange to route rows from a producer thread to a consumer thread, and
- according to whether the exchange preserves the order of the input rows.

Table 3-6 shows how we classify exchanges based on the number of producer and/or consumer threads. The number of threads within each parallel "zone" of a query plan is the same and is equal to the DOP. Thus, an exchange may have exactly one producer or consumer thread if it is at the beginning or end of a serial zone within the plan or it may have exactly DOP producer or consumer threads.

Table 3-6. Types of Parallelism Exchange Operators

Type	# Producer Threads	# Consumer Threads	Gather Streams	DOP	Repartition
Streams	DOP	DOP	Distribute Streams	1	DOP

A gather streams exchange is often called a start parallelism exchange because the operators above it run serially while the operators below it run in parallel. The root exchange in any parallel plan is always a gather exchange since the results of any query plan must ultimately be funneled back to the single connection thread to be returned to the client. A distribute streams exchange is often called a stop parallelism exchange. It is the opposite of a gather streams exchange. The operators above a distribute streams exchange run in parallel, while the operators below it run serially.

Table 3-7 shows how we classify exchanges based on the type of partitioning. Partitioning type only makes sense for a repartition or a distribute streams exchange. There is only one way to route rows in a gather exchange: to the single consumer thread.

Table 3-7. Types of Partitioning for Executing Parallel Queries

Partitioning Type **Description**
Broadcast Send all rows to all consumer threads.
Hash Determine where to send each row by evaluating a hash function on one or more columns in the row.
Round Robin Send each packet of rows to the next consumer thread in sequence.
Demand Send the next row to the next consumer that asks for a row. This partition type is the only type of exchange that uses a pull rather than a push model for data flow. Demand partitioning is used only to distribute partition ids in parallel plans with partitioned tables.
Range Determine where to send each row by evaluating a range function on one column in the row. Range partitioning is used only by certain parallel index build and statistics-gathering plans.

Finally, exchanges can be broken down into merging (or order preserving) and nonmerging (or non-order-preserving) exchanges. The consumer in a merging exchange ensures that rows from multiple producers are returned in a sorted order. (The rows must already be in this sorted order at the producer; the merging exchange does not actually sort.) A merging exchange only makes sense for a gather or a repartition streams exchange; with a distribute streams exchange, there is only one producer and, thus, only one stream of rows and nothing to merge at each consumer.

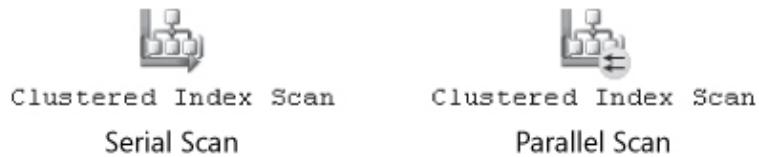
SQL Server commonly uses merging exchanges in plans for queries with ORDER BY clauses or in plans with parallel stream aggregate or merge join operators. However, merging exchanges are generally more expensive than, and do not scale as well as, nonmerging exchanges. A merging exchange cannot return rows until it has received rows from every input; even then, a merging exchange must return rows in sorted order. If some inputs to a merging exchange include rows that are much later in the sort order than rows from the other inputs, the exchange cannot return the rows that sort later. The inputs returning the rows that sort later may get too far ahead of the other inputs and be forced to wait while the other inputs "catch up." The higher the DOP (that is, the more threads), the more likely that some of the inputs will be forced to wait for other inputs.

In some extreme cases, merging exchanges may lead to intra-query "parallel deadlocks." For a query plan to be susceptible to a parallel deadlock, it must include at least two merging exchanges separated by order-preserving operators or a parallel merge join above a pair of merging exchanges. Although parallel deadlocks do not cause a query to fail, they can lead to serious performance degradations. Parallel deadlocks were a far more common problem in SQL Server 2000 than in SQL Server 2005. Improvements in SQL Server 2005 have resolved this problem in all but a few very rare scenarios. To check for a parallel deadlock, use the sys.dm_os_waiting_tasks DMV. If all threads associated with a session are blocked on the CXPACKET wait type, you have a parallel deadlock. Generally, the only solution to a parallel deadlock is to reduce the DOP, force a serial plan, or alter the query plan to eliminate the merging exchanges.

In light of these performance issues, the optimizer costs parallel plans with merging exchanges fairly high and tries to avoid choosing a plan with merging exchanges as the DOP increases.

SQL Server includes all of the above properties in all three query plan varieties (graphical, text, and XML). Moreover, in graphical query plans you can also tell at a glance which operators are running in parallel (that is, which operators are between a start exchange and a stop exchange) by looking for a little parallelism symbol on the operator icons, as illustrated in [Figure 3-25](#).

Figure 3-25. Clustered Index Scan operator icon with and without parallelism



In the remainder of this subsection, we will look at how a few of the operators that we've already discussed earlier in this chapter behave in parallel plans. Since most operators neither need to know nor care whether they are executing in parallel and can be parallelized merely by placing them below a start exchange, we will focus primarily on the handful of operators whose behavior is interesting in the context of parallelism.

Parallel Scan

The scan operator is one of the few operators that is parallel "aware." The threads that compose a parallel scan work together to scan all of the rows in a table. There is no a priori assignment of rows or pages to a particular thread. Instead, the storage engine dynamically hands out pages or ranges of rows to threads. A parallel page supplier coordinates access to the pages or rows of the table. The parallel page supplier ensures that each page or range of rows is assigned to exactly one thread and, thus, is processed exactly once.

At the beginning of a parallel scan, each thread requests a set of pages or a range of rows from the parallel page supplier. The threads then begin processing their assigned pages or rows and begin returning results. When a thread finishes with its assigned set of pages, it requests the next set of pages or the next range of rows from the parallel page supplier.

This algorithm has a couple of advantages:

1. It is independent of the number of threads. SQL Server can add and remove threads from a parallel scan, and it automatically adjusts. If the number of threads doubles, each thread processes (approximately) half as many pages. And, if the I/O system can keep up, the scan runs twice as fast.
2. It is resilient to skew or load imbalances. If one thread runs slower than the other threads, that thread simply requests fewer pages while the other faster threads pick up the extra work. The total execution time degrades smoothly. Compare this scenario to what would happen if SQL Server statically assigned pages to threads: The slow thread would dominate the total execution time.

Let's begin with a simple example. To get parallel plans, we need fairly big tables; if the tables are too small, the optimizer concludes that a serial plan is perfectly adequate. The following script creates two tables. Each table has 250,000 rows and, thanks to the fixed-length char(200) column, well over 6,500 pages.

Code View:

```
CREATE TABLE [HugeTable1]
(
    [Key] int,
    [Data] int,
    [Pad] char(200),
    CONSTRAINT [PK1] PRIMARY KEY ([Key])
)
```

```

SET NOCOUNT ON
DECLARE @i int
BEGIN TRAN
SET @i = 0
WHILE @i < 250000
BEGIN
    INSERT [HugeTable1] VALUES(@i, @i, NULL)
    SET @i = @i + 1
    IF @i % 1000 = 0
    BEGIN
        COMMIT TRAN
        BEGIN TRAN
    END
END
COMMIT TRAN

SELECT [Key], [Data], [Pad]
INTO [HugeTable2]
FROM [HugeTable1]

ALTER TABLE [HugeTable2]
ADD CONSTRAINT [PK2] PRIMARY KEY ([Key])

```

Now let's try the simplest possible query:

```

SELECT [Key], [Data]
FROM [HugeTable1]

```

Despite the large table, this query results in a serial plan:

```
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```

We do not get a parallel plan because parallelism is really about speeding up queries by applying more CPUs to the problem. The cost of this query is dominated by the cost of reading pages from disk (which is mitigated by readahead rather than parallelism) and returning rows to the client. The query uses relatively few CPU cycles and, in fact, would probably run slower if SQL Server parallelized it.

Now suppose we add a fairly selective predicate to the query:

```

SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000

```

This query results in a parallel plan:

```
|--Parallelism(Gather Streams)
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
 WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)))
```

Since we do not have an index on the Data column, SQL Server cannot perform an index seek and must evaluate the predicate for each row. By running this query in parallel, SQL Server distributes the cost of evaluating the predicate across multiple CPUs. (In this case, the predicate is so cheap that it probably does not make much difference whether or not the query runs in parallel.)

Note that the exchange in this plan does not return the rows in any particular order. It simply returns the rows in whatever order it receives them from its producer threads. Now observe what happens if we add an ORDER BY clause to the query:

```
SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000
ORDER BY [Key]
```

The optimizer recognizes that we have a clustered index that can return rows sorted by the Key column. To exploit this index and avoid an explicit sort, the optimizer adds a merging or order-preserving exchange to the plan:

Code View:

```
--Parallelism(Gather Streams, )
--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
 WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
```

We can identify the merging exchange by the ORDER BY clause in the query plan. This clause indicates the order in which the exchange returns the rows.

Load Balancing

As we noted previously, the parallel scan algorithm dynamically allocates pages to threads. We can devise an experiment to see this effect in action. Consider this query:

```
SELECT MIN([Data]) FROM [HugeTable1]
```

This query scans the entire table, but because of the aggregate it uses a parallel plan. The aggregate also ensures that the query returns only one row. Without it, the execution time of this query would be dominated by the cost of returning rows to the client. This overhead could alter the results of our experiment.

Code View:

```
--Stream Aggregate(DEFINE:([Expr1003]=MIN([partialagg1004])))
--Parallelism(Gather Streams)
--Stream Aggregate(DEFINE:([partialagg1004]=MIN([HugeTable1].[Data])))
--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```

Before proceeding with the experiment, notice that this plan includes two stream aggregates instead of the usual one in a serial plan. We refer to the bottommost aggregate as a local or partial aggregate. The local

aggregate computes the minimum value of the Data column within each thread. We refer to the topmost aggregate as a global aggregate. The global aggregate computes the minimum of the local minimums. Local aggregation improves query performance in two ways. First, it reduces the number of rows flowing through the exchange. Second, it enables parallel execution of the aggregate. Since this query computes a single result row (recall that it is a scalar aggregate), without the local aggregate, all of the scanned rows would have to be processed by a single thread. SQL Server can use local aggregation for both stream and hash aggregates, for most built-in and many user-defined aggregate functions, and for both scalar aggregates (as in this example) and grouping aggregates. In the case of grouping aggregates, the benefits of local aggregation increase as the number of groups decreases. The smaller the number of groups, the more work the local aggregate is actually able to perform.

Returning to our experiment, using SET STATISTICS XML ON we can run the above query and see exactly how many rows each thread processes. (The same information is also visible with graphical showplan in the Properties window.) Here is an excerpt of the XML output on a two-processor system:

Code View:

```
<RelOp NodeId="3" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index Scan" ...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="1" ActualRows="124986" ... />
    <RunTimeCountersPerThread Thread="2" ActualRows="125014" ... />
    <RunTimeCountersPerThread Thread="0" ActualRows="0" ... />
  </RunTimeInformation>
</RelOp>
```

We can see that both threads (threads 1 and 2) processed approximately half of the rows. (Thread 0 is the coordinator or main thread. It only executes the portion of the query plan above the topmost exchange. Thus, we do not expect it to process any rows for operators executed in a parallel portion of the query plan.)

Now let's repeat the experiment, but this time let's run an expensive serial query at the same time. This cross join query will run for a very long time and use plenty of CPU cycles. The OPTION (MAXDOP 1) query hint forces SQL Server to execute this query in serial. We will discuss this and other hints in more detail in [Chapters 4 and 5](#).

```
SELECT MIN(T1.[Key] + T2.[Key])
FROM [HugeTable1] T1 CROSS JOIN [HugeTable2] T2
OPTION (MAXDOP 1)
```

This serial query runs with a single thread and consumes cycles from only one of the two schedulers. While it is running, we run the parallel scan query again. Here is the statistics XML output for the second run:

Code View:

```
<RelOp NodeId="3" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index Scan" ...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="1" ActualRows="54232" ... />
    <RunTimeCountersPerThread Thread="2" ActualRows="195768" ... />
    <RunTimeCountersPerThread Thread="0" ActualRows="0" ... />
  </RunTimeInformation>
</RelOp>
```

This time thread 1 processed over 75 percent of the rows while thread 2, which was busy executing the serial plan, processed fewer than 25 percent of the rows. The parallel scan automatically balanced the work across the two threads. Because thread 1 had more free cycles (it wasn't competing with the serial plan), it requested and scanned more pages.

Caution



If you try this experiment, don't forget to terminate the serial query when you are done! Otherwise, it will continue to run and waste cycles for a very long time.

The same load balancing that we just observed applies equally whether a thread is slowed down because of an external factor (such as the serial query in this example) or because of an internal factor. For example, if it costs more to process some rows than others, we will see the same behavior.

Parallel Nested Loops Join

SQL Server parallelizes a nested loops join by distributing the outer rows (that is, the rows from the first input) randomly among the nested loops join threads. For example, if there are two threads running a nested loops join, SQL Server sends about half of the rows to each thread. Each thread then runs the inner side (that is, the second input) of the nested loops join for its set of rows as if it were running serially. That is, for each outer row assigned to it, the thread executes its inner input using that row as the source of any correlated parameters. In this way, the threads can run independently. SQL Server does not add exchanges to or parallelize the inner side of a nested loops join.

Here is a simple example of a parallel nested loops join:

```
SELECT T1.[Key], T1.[Data], T2.[Data]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
    ON T1.[Key] = T2.[Key]
WHERE T1.[Data] < 100
```

Let's analyze the STATISTICS PROFILE output for this plan:

Code View:

```
Rows Executes
100   1    |--Parallelism(Gather Streams)
100   2    |--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key]) OPTIMIZED)
100   2      |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]),
           WHERE:([T1].[Data]<(100)))
100  100   |--Clustered Index Seek (OBJECT:([HugeTable2].[PK2] AS [T2]),
           SEEK:([T2].[Key]=[T1].[Key])
           ORDERED FORWARD)
```

Note that there is only one exchange (in other words, parallelism operator) in this plan. This exchange is at the root of the query plan, therefore all of the operators in this plan (the nested loops join, the table scan, and the clustered index seek) execute in each thread. The lack of an index on the Data column of HugeTable1 forces the clustered index scan and the use of a residual predicate to evaluate T1.[Data] < 100. Because there are 250,000 rows in HugeTable1, the scan is expensive and, as in the previous examples, the optimizer chooses a

parallel scan of HugeTable1. The scan of HugeTable1 is not immediately below an exchange (in other words, a parallelism operator). In fact, it is on the outer side of the nested loops join, and it is the nested loops join that is below the exchange. Nevertheless, because the scan is on the outer side of the join and because the join is below a start (in other words, a gather) exchange, SQL Server performs a parallel scan of HugeTable1. Since a parallel scan assigns pages to threads dynamically, the scan distributes the HugeTable1 rows among the threads. It does not matter which rows it distributes to which threads.

Because SQL Server ran this query with DOP 2, we see that there are two executes each for the clustered index scan of HugeTable1 and for the join (both of which are in the same thread). Moreover, the scan and join both return a total of 100 rows, though we cannot tell from this output how many rows each thread returned. We can (and momentarily will) determine this information using statistics XML output.

Next, the join executes its inner side (in this case, the clustered index seek on HugeTable2) for each of the 100 outer rows. Here is where things get a little tricky. Although each of the two threads contains an instance of the clustered index seek iterator, and even though the seek is below the join, which is below the exchange, the seek is on the inner side of the join and, thus, the seek does not use a parallel scan. Instead, the two seek instances execute independently of one another on two different outer rows and two different correlated parameters. As in a serial plan, we see 100 executes of the index seek: one for each row on the outer side of the join. With only one exception, which we will address below, no matter how complex the inner side of a nested loops join is, we always execute it as a serial plan just as in this simple example.

Round-Robin Exchange

In the previous example, SQL Server relies on the parallel scan to distribute rows uniformly among the threads. In some cases, SQL Server must add a round-robin exchange to distribute the rows. (Recall that a round-robin exchange sends each subsequent packet of rows to the next consumer thread in a fixed sequence.) Here is one such example:

```
SELECT T1_Top.[Key], T1_Top.[Data], T2.[Data]
FROM
(
    SELECT TOP 100 T1.[Key], T1.[Data]
    FROM [HugeTable1] T1
    ORDER BY T1.[Data]
) T1_Top,
[HugeTable2] T2
WHERE T1_Top.[Key] = T2.[Key]
```

Here is the corresponding plan:

Code View:

```
--Parallelism(Gather Streams)
|--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key],
[Expr1004]) WITH UNORDERED PREFETCH)
|--
|   |--Top(TOP EXPRESSION:((100)))
|       |--Parallelism(Gather Streams, ORDER BY:([T1].[Data] ASC))
|           |--Sort(TOP 100, ORDER BY:([T1].[Data] ASC))
|               |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]))
|--Clustered Index Seek(OBJECT:([HugeTable2].[PK2] AS [T2]),
SEEK:([T2].[Key]=[T1].[Key]) ORDERED FORWARD)
```

The main difference between this plan and the plan from the original example is that this plan includes a top-100 rows iterator. The top iterator can only be correctly evaluated in a serial plan thread. (It cannot be split among multiple threads or we might end up with too few or too many rows.) Thus, SQL Server must add a stop (that is, a distribute streams) exchange above the top iterator and cannot use the parallel scan to distribute the rows among the join threads. Instead SQL Server parallelizes the join by having the stop exchange use round-robin partitioning to distribute the rows among the join threads.

Parallel Nested Loops Join Performance

The parallel scan has one major advantage over the round-robin exchange. A parallel scan automatically and dynamically balances the workload among the threads, while a round-robin exchange does not. As the previous parallel scan example demonstrates, if we have a query in which one thread is slower than the others, the parallel scan may help compensate.

Both the parallel scan and a round-robin exchange may fail to keep all join threads busy if there are too many threads and too few pages and/or rows to be processed. Some threads may get no rows to process and end up idle. This problem can be more pronounced with a parallel scan since it doles out multiple pages at one time to each thread while the exchange distributes one packet (equivalent to one page) of rows at one time.

We can see this problem in the original parallel nested loops join example above by checking the statistics XML output:

```
<RelOp NodeId="1" PhysicalOp="Nested Loops" LogicalOp="Inner Join" ...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="2" ActualRows="0" ... />
    <RunTimeCountersPerThread Thread="1" ActualRows="100" ... />
    <RunTimeCountersPerThread Thread="0" ActualRows="0" ... />
  </RunTimeInformation>
</RelOp>
```

This output shows that all of the join's output rows are processed by thread 1. The problem is that the clustered index scan of HugeTable1 has a residual predicate $T1.[Data] < 100$, which is true for the first 100 rows in the table and false for the remaining rows. The roughly three pages containing the first 100 rows are all assigned to the first thread.

In this example, this is not a big problem since the inner side of the join is fairly cheap and contributes a small percentage of the overall cost of the query (with the clustered index scan of HugeTable1 itself contributing a much larger percentage of the cost). However, this problem could be more significant if the inner side of the query were more expensive. The problem is especially notable in SQL Server 2005 with partitioned tables which, as noted earlier, use nested loops joins to enumerate partition ids. For instance, if we try to perform a parallel scan of two partitions of a large table, the DOP of the scan is effectively limited to two because of the nested loops join.

Inner-side Parallel Execution

We noted above that, with one exception, SQL Server always executes the inner side of a parallel nested loops join as a serial plan. The exception occurs when the optimizer knows that the outer input to the nested loops join is guaranteed to return only a single row and when the join has no correlated parameters. That is, the inner side of the join is guaranteed to run exactly once and can be run independently of the outer side of the join. In this case, if the join is an inner join, the inner side of the join may include a parallel scan. Moreover, in some rare scenarios, if the join is an outer or semi-join, it may be executed in a serial zone (below a stop exchange) and the inner side of the join may include exchanges.

For example, consider this query:

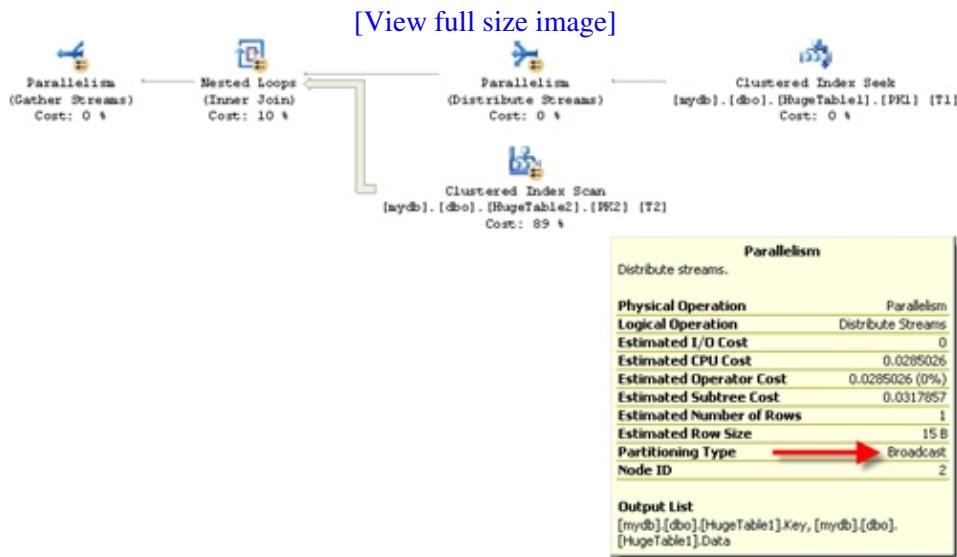
```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
WHERE T1.[Key] = 0
```

This query yields the following plan:

```
--Parallelism(Gather Streams)
|--Nested Loops(Inner Join, WHERE:([T1].[Data]=[T2].[Data]))
|  |--Parallelism(Distribute Streams, Broadcast Partitioning)
|    |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
|      SEEK:([T1].[Key]=(0)) ORDERED FORWARD)
|  |--Clustered Index Scan(OBJECT:([HugeTable2].[PK2] AS [T2]))
```

The equality predicate `T1.[Key] = 0` ensures that the clustered index seek of HugeTable1 returns exactly one row. Notice that the plan includes a broadcast exchange, which delivers this single row to each instance of the nested loops join. In this plan, the clustered index scan of HugeTable2 uses a parallel scan. That is, all instances of this scan cooperatively scan HugeTable2 exactly once. If this plan used a serial scan of HugeTable2, each scan instance would return the entire contents of HugeTable2 which, in conjunction with the broadcast exchange, would result in the plan returning each row multiple times. [Figure 3-26](#) shows the graphical plan for this query, along with the ToolTip indicating that a broadcast exchange is being used.

[Figure 3-26. A parallel plan using a broadcast exchange](#)



Parallel Merge Join

SQL Server parallelizes merge joins by distributing both sets of input rows among the individual merge join threads using hash partitioning. If two input rows join, they have the same values for the join key and, therefore, hash to the same merge join thread. Unlike the parallel plan examples we've seen so far, merge join also requires that its input rows be sorted. In a parallel merge join plan, as in a serial merge join plan, SQL

Server can use an index scan to deliver rows to the merge join in the correct sort order. However, in a parallel plan, SQL Server must also use a merging exchange to preserve the order of the input rows.

Although, as we discussed previously, the optimizer tends to favor plans that do not require a merging exchange; by including an ORDER BY clause in a join query, we can encourage such a plan. For example, consider the following query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
    ON T1.[Key] = T2.[Data]
ORDER BY T1.[Key]
```

The optimizer chooses the following query plan:

Code View:

```
--Parallelism(Gather Streams, ORDER BY:([T1].[Key] ASC))
--Merge Join(Inner Join, MERGE:([T1].[Key])=([T2].[Data]), RESIDUAL:(...))
  |--Parallelism(Repartition Streams, Hash Partitioning,
    PARTITION COLUMNS:([T1].[Key]), ORDER BY:([T1].[Key] ASC))
  |  |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]), ORDERED FORWARD)
  |--Sort(ORDER BY:([T2].[Data] ASC))
    |--Parallelism(Repartition Streams, Hash Partitioning,
      PARTITION COLUMNS:([T2].[Data]))
    |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

This plan includes three exchanges including two merging exchanges. Two of the exchanges are hash partitioning exchanges and are placed below the merge join. These exchanges ensure that any input rows that might potentially join are delivered to the same merge join thread. The first merge join input uses a merging exchange, and the clustered index scan of HugeTable1 delivers the input rows sorted on the join key. The second merge join input uses a nonmerging exchange. The clustered index scan of HugeTable2 does not deliver input rows sorted on the join key. Instead there is a sort between the merge join and the exchange. By placing the exchange below the sort, the optimizer avoids the need to use a merging exchange. The final exchange is the start exchange at the top of the plan. This exchange returns the rows in the order required by the ORDER BY clause on the query and takes advantage of the property that the merge join output rows in the same order as it inputs them.

Parallel Hash Join

SQL Server uses one of two different strategies to parallelize a hash join. One strategy uses hash partitioning just like a parallel merge join; the other strategy uses broadcast partitioning and is often called a broadcast hash join.

Hash Partitioning

The more common strategy for parallelizing a hash join involves distributing the build rows (in other words, the rows from the first input) and the probe rows (in other words, the rows from the second input) among the individual hash join threads using hash partitioning. As in the parallel merge join case, if two input rows join, they are guaranteed to hash to the same hash join thread. After the data has been hash partitioned among the threads, the hash join instances all run completely independently on their respective data sets. Unlike merge join, hash join does not require that input rows be delivered in any particular order and, thus, it does not

require merging exchanges. The absence of any interthread dependencies ensures that this strategy scales extremely well as the DOP increases.

To see an example of a parallel hash join, consider the following simple query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
```

SQL Server executes this query using the following plan:

Code View:

```
|--Parallelism(Gather Streams)
 |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
   |--Parallelism(Repartition Streams, Hash Partitioning,
     PARTITION COLUMNS:([T1].[Data]))
   |  |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]))
   |--Parallelism(Repartition Streams, Hash Partitioning,
     PARTITION COLUMNS:([T2].[Data]))
   |  |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

Broadcast Partitioning

Consider what happens if SQL Server tries to parallelize a hash join using hash partitioning, but there are only a small number of rows on the build side of the hash join. If there are fewer rows than hash join threads, some threads might receive no rows at all. In this case, those threads would have no work to do during the probe phase of the join and would remain idle. Even if there are more rows than threads, the presence of duplicate key values and/or skew in the hash function means some threads might receive and process many more rows than other threads.

To reduce the risk of skew problems, when the optimizer estimates that the number of build rows is relatively small, it may choose to broadcast these rows to all of the hash join threads. Since all build rows are broadcast to all hash join threads, in a broadcast hash join, it does not matter which threads process which probe rows. Each probe row can be sent to any thread and, if it can join with any build rows, it will.

For example, the following query includes a very selective predicate:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
```

Because of the selective predicate, the optimizer chooses a broadcast hash join plan:

Code View:

```
|--Parallelism(Gather Streams)
 |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
   |--Parallelism(Distribute Streams, )
   |  |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
     SEEK:([T1].[Key] < (100)) ORDERED FORWARD)
   |--Clustered Index Scan(OBJECT:([HugeTable2].[PK2] AS [T2]))
```

Note that the exchange above the clustered index seek of HugeTable1 is now a broadcast exchange, while the exchange above the clustered index scan of HugeTable2 is gone. This plan does not need an exchange above the scan of HugeTable2 because the parallel scan automatically distributes the pages and rows of HugeTable2 among the hash join threads. This result is similar to how the parallel scan distributed rows among nested loops join threads for the parallel nested loops join, which we discussed earlier. Similar to the parallel nested loops join, if there is a serial zone on the probe input of a broadcast hash join (for example, caused by a top operator), the query may use a round-robin exchange to redistribute the rows.

While broadcast hash joins do reduce the risk of skew problems, they are not suitable for all scenarios. In particular, broadcast hash joins use more memory than their hash-partitioned counterparts. Because a broadcast hash join means SQL Server sends every build row to every hash join thread, if the number of threads doubles, the amount of memory consumed also doubles. Thus, a broadcast hash join requires memory that is proportional to the degree of parallelism, whereas a hash-partitioned parallel hash join requires the same amount of memory regardless of the degree of parallelism.

Bitmap Filtering

Next, suppose we have a moderately selective predicate on the build input to a hash join:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
```

The predicate `T1.[Key] < 10000` is not selective enough to generate a broadcast hash join. This predicate does, however, eliminate 96 percent of the build rows from HugeTable1. It also indirectly eliminates 96 percent of the rows from HugeTable2 because they no longer join with rows from HugeTable1. It would be nice if there was a way to eliminate these rows from HugeTable2 much earlier without the overhead of passing the rows through the exchange and into the hash join. The bitmap operator provides just such a mechanism:

Code View:

```
--Parallelism(Gather Streams)
  |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
    |--Parallelism(Repartition Streams, Hash Partitioning,
      PARTITION COLUMNS:([T1].[Data]))
        |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
          SEEK:([T1].[Key] < (10000)) ORDERED FORWARD)
    |--Parallelism(Repartition Streams, Hash Partitioning,
      PARTITION COLUMNS:([T2].[Data]), )
        |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

As its name suggests, the bitmap operator builds a bitmap. Just like the hash join, the bitmap operator hashes each row of HugeTable1 on the join key and sets the corresponding bit in the bitmap. Once the scan of HugeTable1 and the hash join build are complete, SQL Server transfers the bitmap to the exchange above HugeTable2. This exchange uses the bitmap as a filter; notice the WHERE clause in the plan. The exchange hashes each row of HugeTable2 on the join key and tests the corresponding bit in the bitmap. If the bit is set,

the row may join and the exchange passes it along to the hash join. If the bit is not set, the row cannot join and the exchange discards it.

Although it is somewhat rarer, some parallel merge joins also use a bitmap operator. To use the bitmap operator, a merge join must have a sort on its left input. The optimizer adds the bitmap operator below the sort. Recall that a merge join processes both inputs concurrently. Without the sort, there would be no way to build the bitmap on one input before beginning to process the other input. However, because the sort is blocking, the plan can build a bitmap on the left input before the merge join begins processing its right input.

Inserts, Updates, and Deletes

Data Modification (INSERT, UPDATE, and DELETE) statements could be the subject of an entire chapter. In this section, we provide just a brief overview of data modification statement plans. These plans consist of two sections: a "read cursor" and a "write cursor." The read cursor determines which rows will be affected by the data modification statement. The read cursor works like a normal SELECT statement. All of the material about understanding queries that we have discussed in this chapter, including the discussion of parallelism, applies equally well to the read cursor of a data modification plan. The write cursor executes the actual INSERTS, UPDATES, or DELETES. The write cursor, which can get extremely complex, also executes other side effects of the data modification such as nonclustered index maintenance, indexed view maintenance, referential integrity constraint validation, and cascading actions. The write cursor is implemented using many of the same operators that we've already discussed. However, unlike the read cursor, where we can often impact performance by rewriting the query or by using hints (which we will discuss in [Chapters 4 and 5](#)), we have comparatively little control over the write cursor, query plan and performance. Short of creating or dropping indexes or constraints, we cannot control the list of indexes that must be maintained and constraints that must be validated. It is also worth noting that the write cursor is never executed using parallelism.

For example, consider the following UPDATE statement, which changes the shipper for orders shipped to London. This statement is guaranteed to violate the foreign key constraint on the ShipVia column and will fail.

```
UPDATE [Orders]
SET [ShipVia] = 4
WHERE [ShipCity] = N'London'
```

This statement uses the following query plan:

```
--Assert(WHERE:(CASE WHEN [Expr1023] IS NULL THEN (0) ELSE NULL END))
--Nested Loops(Left Semi Join, OUTER REFERENCES:([Orders].[ShipVia]),
DEFINE:([Expr1023] = [PROBE VALUE]))
--Clustered Index Update(OBJECT:([Orders].[PK_Orders]),
OBJECT:([Orders].[ShippersOrders]),
SET:([Orders].[ShipVia] = [Expr1019]))
| --Compute Scalar(DEFINE:([Expr1021]=[Expr1021]))
| --Compute Scalar(DEFINE:([Expr1021]=CASE WHEN [Expr1003]
THEN (1)
ELSE (0) END))
| --Compute Scalar(DEFINE:([Expr1019]=(4)))
| --Compute Scalar(DEFINE:([Expr1003]=
CASE WHEN [Orders].[ShipVia] = (4)
THEN (1) ELSE (0) END))
| --Top(ROWCOUNT est 0)
| --Clustered Index Scan
    (OBJECT:([Orders].[PK_Orders]),
    WHERE:([Orders].[ShipCity]=N'London') ORDERED)
--Clustered Index Seek(OBJECT:([Shippers].[PK_Shippers]),
SEEK:([Shippers].[ShipperID]=[Orders].[ShipVia]) ORDERED FORWARD)
```

The read cursor for this plan consists solely of the clustered index scan of the Orders table, whereas the write cursor consists of the entire remainder of the plan. The write cursor includes a clustered index update operator, which updates two indexes on the Orders table— the PK_Orders clustered index and the ShippersOrders nonclustered index. The write cursor also includes a join with the Shippers table and an assert operator, which validate the foreign key constraint on the ShipVia column.

Summary

In this chapter, we've explained the basics of query processing in SQL Server. We've introduced iterators—the fundamental building blocks of query processing—and discussed how to view query plans in three different formats: text, graphical, and XML. We've covered many of the most common iterators, including the iterators that implement scans and seeks, joins, and aggregation. We've looked at more complex query plans including dynamic index seeks, index unions and intersection, and a variety of subquery examples. We've also explored how the query processor implements parallel query execution. All of the query plans that SQL Server supports are built from these and other basic operators.

The vast majority of the time, the SQL Server query processor works extremely well. The optimizer generally chooses satisfactory query plans, and most applications work well without any special tuning. However, there will be times when your queries are not performing optimally and in the next chapter we'll discuss options for detecting and correcting problems. The more you understand about what actually happens during query execution, the better you'll be able to detect when a plan is not optimal, and when there is room for improvement.

Chapter 4. Troubleshooting Query Performance

â Kalen Delaney and Craig Freedman

In this chapter:	
Compilation and Optimization	199
Detecting Problems in Plans	218
Monitoring Query Performance	221
Query Improvements	225
Query Processing Best Practices	273
Summary	275

As mentioned in [Chapter 3](#), "Query Execution," the SQL Server query processor consists of two components: the query optimizer and the query execution engine. In [Chapter 3](#), we looked at details of many different types of query plans and discussed the process of query execution. In the first part of this chapter, we'll talk about the query optimizer, which is responsible for generating good query plans. The optimizer includes subcomponents to gather statistics, perform cardinality estimation using these statistics, estimate the cost of query plans, and, of course, explore alternative query plans in order to generate a good plan for execution. In the second part of the chapter, we'll discuss techniques to detect suboptimal query plans. The third section describes techniques for improving your query's performance and includes a discussion of the new query optimization hints introduced in SQL Server 2005. The chapter ends with a list of best-practice guidelines to

keep in mind when building any SQL Server-based application.

Compilation and Optimization

It's important to be aware that the compilation and optimization process is completely separate from the process of execution. The gap between when SQL Server compiles a query and when the query is executed can be as short as a few microseconds or as long as several days. Therefore, during compilation (which includes optimization), SQL Server has to determine what kind of available information will be useful when the query is executed. Not everything that is true at compilation time will be true at execution time. For example, if the optimizer took into account how many concurrent users were trying to access the same data as the query being optimized, that information could change dramatically by the time the query was executed. Even the amount of data and the data distribution can change between compilation and execution, so we really need to consider the compilation phase of query processing completely separately from the execution phase.

Compilation

When a query is ready to be processed by SQL Server, an execution plan for the query might already be available in SQL Server's plan cache. If no usable plan is available, the query must be compiled. The methods by which execution plans can be saved and reused are discussed in detail in [Chapter 5, "Plan Caching and Recompilation."](#) In this section, we're assuming there is no available plan and your query must be compiled. The compilation process encompasses a few steps. First, your query is processed by the parser and then by a component called the algebrizer. The parser handles the process of dissecting and transforming your SQL statements into compiler-ready data structures. Parsing also includes validating the syntax to ensure that it is legal. If the parser determines that your SQL statement is a DML statement, it creates an expression parse tree. For non-DML statements, such as DDL for object creation statements, the parser produces a sequence tree. The sequence tree is one of the few data structures in SQL Server 2005 that has survived from SQL Server 6.5. The main difference between a sequence tree and a parse tree is that the sequence tree is a binary tree, whereas the parse tree can have multiple children attached to each node. In addition, some DDL statements need to include a parse tree as well as a sequence tree. For example, in a CREATE VIEW statement, the SELECT statement comprising the view definition is parsed as an expression tree while the wrapper is parsed as a sequence tree. Also, if a CREATE TABLE statement includes default values or check constraints, those get parsed as expression trees.

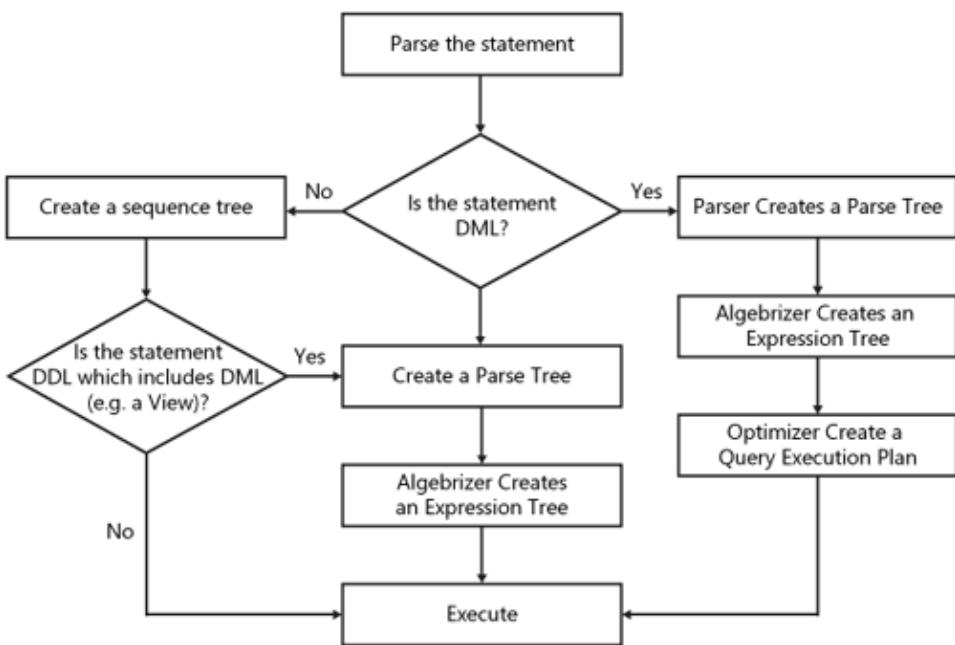
Parsing doesn't include things such as checking for valid table and column names, which are dealt with by the algebrizer. The algebrizer basically determines the characteristics of the objects that you reference inside your SQL statements and checks whether the semantics you're asking for make sense. For example, it is semantically illogical to try to execute a table. In addition, the algebrizer makes sure your objects are valid, including verification that the column names referenced actually do exist in the referenced tables and binding the object IDs and column IDs to the object references. The algebrizer only acts on expression parse trees (not on sequence trees) and its output is an algebrized expression tree.

The algebrized expression tree is then compiled by the query optimizer. After the execution plan is generated, it is placed in cache and then executed.

[Figure 4-1](#) shows the flow of this compilation process, leading up to query execution.

Figure 4-1. The steps in compiling a statement

[\[View full size image\]](#)



Optimization

SQL Server's query optimizer is a cost-based optimizer, which means that it tries to come up with a low-cost execution plan for each SQL statement. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The query optimizer must analyze possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it tries to find an execution plan that has a cost reasonably close to the theoretical minimum. Later in this section, we'll discuss some of the ways that the optimizer can reduce the amount of time it needs to spend on optimization.

The lowest estimated cost is not simply the lowest resource cost; the query optimizer chooses a plan that balances the amount of time needed to return results to the user with a reasonable cost in resources. For example, processing a query in parallel (using multiple CPUs simultaneously for the same query) typically uses more resources than processing it serially using a single CPU, but the query completes much faster. The optimizer will choose a parallel execution plan to return results if the load on the server will not be adversely affected.

Optimization itself involves many steps. The first step is called trivial plan optimization. The idea behind trivial plan optimization is that cost-based optimization is expensive to run. The optimizer can try many possible variations in looking for a cheap plan. If SQL Server knows that there is only one really viable plan for a query, it can avoid a lot of work. A common example is a query that consists of an INSERT with a VALUES clause. There is only one possible plan. Another example is a SELECT statement, in which the columns in the WHERE clause include all keys of a unique index and that index is the only one that is relevant; that is, no other index has that set of columns in it. In these two cases, SQL Server should just generate the plan and not try to find something better. Trivial plan optimization finds the really obvious plans that are typically very inexpensive. This saves the optimizer from having to consider every possible plan, which can be costly and can outweigh any benefit provided by well-optimized queries. You can determine that the trivial plan optimization has found a plan in one of two ways. You can look at the graphical execution plan, and open up the properties window. Clicking on the left-most iterator will show properties for the entire query. In the left column of the properties list, look for Optimization Level. The right column will show either TRIVIAL or FULL. Alternatively, you can generate an XML plan for a query. Near the top of the XML document, within an element called StmtSimple, there will be an attribute called StatementOptmLevel. A

value of TRIVIAL for this attribute indicates a plan found by the trivial plan optimization.

If the trivial plan optimization doesn't find a simple plan, SQL Server can perform some simplifications, which are usually syntactic transformations of the query itself, to look for commutative properties and operations that can be rearranged. SQL Server can perform operations that don't require looking at the cost or analyzing what indexes are available but that can result in a more efficient query. SQL Server then loads up the metadata, including the statistical information on the indexes, and then the optimizer goes through a series of phases of cost-based optimization.

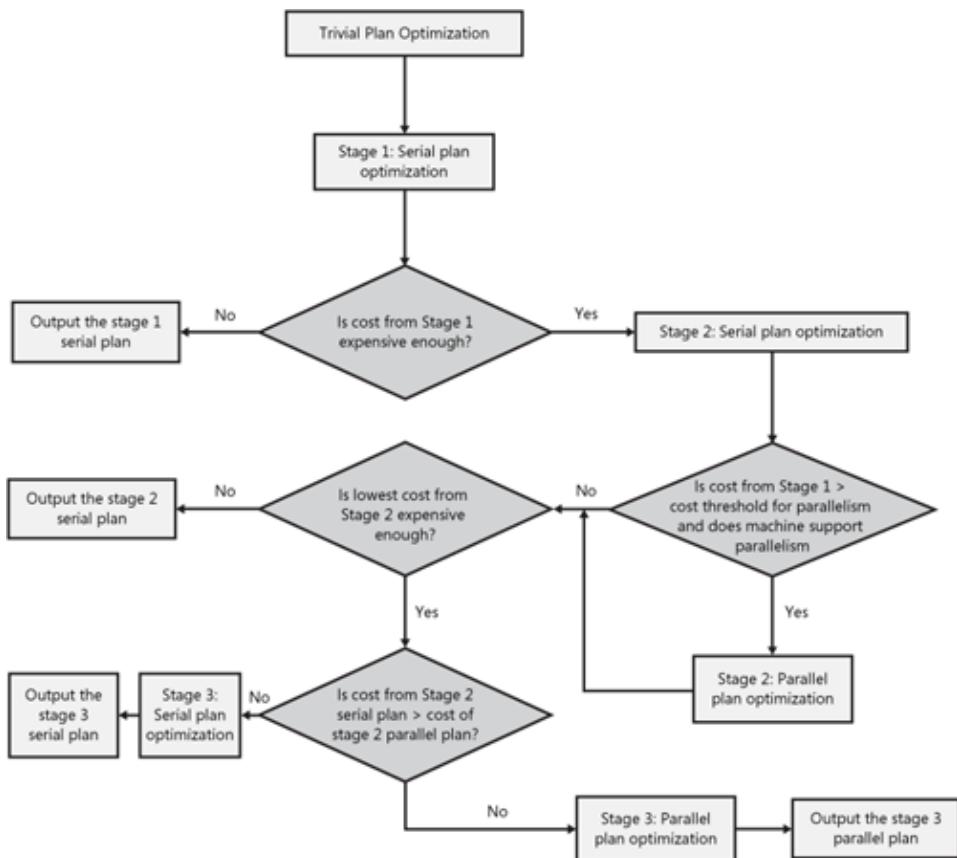
Cost-based optimization is designed as a set of transformation rules that try various permutations of indexes, join strategies, aggregation strategies, table orderings, etc. Because of the number of potential plans in SQL Server 2005, if the optimizer just ran through all the combinations and produced a plan, the optimization process would take a long time. So optimization is broken up into three stages. Each stage is a set of rules. After each stage, SQL Server evaluates the cost of any resulting plan, and if the plan is cheap enough, that plan is considered the final plan. If the plan is not cheap enough, the optimizer runs the next stage, which extends the set of rules.

Stage 1 considers only serial plans (single processor) and if it finds a good enough serial plan, the optimizer does not proceed to the subsequent stages. If the cost from stage 1 is not cheap enough, stage 2 first tries to find a serial plan, and then considers a parallel plan. The parallel plan will only be considered if the machine supports parallelism and is configured to use multiple processors for a single query, and if the configuration option cost threshold for parallelism is less than the cost at the end of stage 1.

If the best cost after stage 2 is still not cheap enough, the optimizer proceeds to stage 3. If the serial plan cost at the end of stage 2 is greater than the parallel plan cost at the end of stage 2, stage 3 will only consider parallel plans. Otherwise the optimizer will only consider serial plans. At the end of stage 3, the cheapest plan is considered the final plan and that is the plan that will be executed. [Figure 4-2](#) shows the flow of processing through the optimizer.

Figure 4-2. The steps in optimizing a statement

[[View full size image](#)]



How the Query Optimizer Works

Monitoring and tuning queries are essential steps in optimizing performance. Knowing how the query optimizer works can be helpful as you think about how to write a good query or what indexes to define. However, you should guard against outsmarting yourself and trying to predict what the optimizer will do. You might miss some good options this way. Try to write your queries in the most intuitive way you can, and then try to tune them only if their performance doesn't seem good enough.

Nevertheless, insight into how the optimizer works is certainly useful. It can help you understand why certain indexes are recommended over others, and it can help you understand the query plan output in SQL Server Management Studio. For each table involved in the query, the query optimizer evaluates the search conditions in the WHERE clause and considers which indexes are available to narrow the scan of a table. That is, the optimizer evaluates to what extent an index can exclude rows from consideration. The more rows that can be excluded, the better, because that leaves fewer rows to process.

As discussed in [Chapter 3](#), SQL Server can process joins, aggregations, unions, etc. in many different ways. For each potential strategy, the optimizer determines which index or indexes might be useful and then estimates a cost, taking into account the number of logical reads and the memory resources that are required and available. The optimizer compares the cost of each potential plan and chooses the plan with the lowest estimate. Prior to determining the useful indexes, the optimizer performs a step that we can call **Query Analysis**, which we'll look at next. After that, we'll examine how SQL Server determines the usefulness of your existing indexes.

Query Analysis

During the query analysis phase, the query optimizer looks at each clause of the query and determines whether it can be useful in limiting how much data must be scanned— that is, whether the clause is useful as a search argument (SARG) or as part of the join criteria. A clause that can be used as a SARG is referred to as sargable, or optimizable, and can make use of an index for faster retrieval.

A SARG limits a search because it specifies an exact match, a range of values, or a conjunction of two or more limiting conditions combined by AND. A SARG contains a constant expression (or a variable that is resolved to a constant) that acts on a column by using an operator. It has the form

```
column inclusive_operator <constant or variable>
```

or

```
<constant or variable> inclusive_operator column
```

The column name can appear on one side of the operator, and the constant or variable can appear on the other side. If a column appears on both sides of the operator, the clause is not sargable. Sargable operators include =, >, <, =>, <=, BETWEEN, and sometimes LIKE. Whether LIKE is sargable depends on the type and position of wildcards used. For example, LIKE 'Jon%' is sargable but LIKE '%Jon' is not because the wildcard (%) at the beginning prevents the use of an index. Here are some SARG examples:

```
name = 'jones'
salary > 40000
60000 < salary
department = 'sales'
name = 'jones' AND salary > 100000
name LIKE 'dail%'
```

A single SARG can include many conditions if they are AND'ed together. That is, one index might be able to operate on all the conditions that are AND'ed together. In the example above, there might be an index on (name, salary), so the entire clause name = 'jones' AND salary > 100000 can be considered one SARG and can be evaluated for qualifying rows using one index. If OR is used instead of AND in this example, a single index scan cannot be used to qualify both terms. The reason should be clear—if the lead field in the index key is name, the index is useful for finding just the 'jones' rows. If the condition in the WHERE clause includes AND salary > 100000, the second field of the index also qualifies those rows. That is, all the rows that have a name value of 'jones' can be further restricted to find the rows with 'jones' and an appropriate salary. The set of rows that meets both of the AND conditions is a subset of the rows that meet just the first condition on name. But if the second criterion is OR salary > 100000, the index is not useful in the same way because all the rows would need to be examined, not just the 'jones' entries. The set of rows that meet either one of the conditions is a superset of the rows that meet just the first condition on name.

A phone book analogy can also be useful. If you want to find people who have the surname "Jones" AND live on 5th Avenue, using the phone book can help greatly reduce the size of your search. But if you want to find people who have the name "Jones" OR live on 5th Avenue, you have to scan every entry in the entire phone book. This assumes that you have only one phone book that is sorted alphabetically by name. If you have two phone books, one sorted by name and one sorted by street, that's another story, which we'll discuss a bit later.

An expression that is not sargable cannot limit the search. That is, SQL Server must evaluate every row to determine whether it meets the conditions in the WHERE clause. So an index is not useful to nonsargable expressions. Typical nonsargable expressions include referencing a column in an expression or function, certain negation operators such as NOT, NOT IN, and NOT LIKE and strings with a wildcard at the beginning. Don't extrapolate too far and think that this means using a nonsargable clause always results in a table scan. An index is not useful to the nonsargable clause, but there might be indexes useful to other SARGs in the query. Queries often have multiple clauses, so a great index for one or more of the other clauses might be available. Here are some examples of nonsargable clauses:

```
ABS(price) < 4
name LIKE '%jon%'
name = 'jones' OR salary > 100000
```

Note



The last example above is not a single search argument, but each expression on either side of the OR is individually sargable. So a single index won't be used to evaluate both expressions, as it might if the operator is AND. A separate index can still be useful to each expression.

Let's look at a specific example. The Northwind2 database includes a table called Order Details, which has a nonclustered index on the column ProductID. When a query contains a SARG involving ProductID, the query optimizer has to determine whether using the nonclustered index will be the fastest way to access the qualifying rows. Remember that a nonclustered index contains every key value in its leaf level, with a bookmark pointing to the actual location of the row having that key. If too many rows satisfy the SARG's condition, the optimizer might decide it's too expensive to use the index and opt instead to scan the entire table. In addition to metadata keeping track of the key columns and properties of each index, SQL Server also maintains statistical information about the distribution of all the different key values in each index. We'll look at statistics in detail a bit later, but for now, be aware that examining the statistics is part of determining whether an index is useful.

The first query below has a SARG involving the CustomerID column in the BigOrders table in the Northwind2 database. SQL Server is able to use the statistical information to estimate that 25 rows will meet the condition in the WHERE clause. The optimizer will choose to use the index on CustomerID.

```
USE Northwind2
SELECT * FROM BigOrders
WHERE CustomerID = 'Ocean';
```

In the next query, because of the presence of the function UPPER, there is no SARG and the index on CustomerID is not even considered. The optimizer has no real choice other than to scan the entire table using a Clustered Index Scan.

```
USE Northwind2
SELECT * FROM BigOrders
WHERE UPPER(CustomerID) = 'OCEAN';
```

Index Selection

During the second phase of query optimization, index selection, the query optimizer determines whether an index exists for a sargable clause, assesses the index's usefulness by determining the selectivity of the clause (that is, how many rows will be returned), and estimates the cost of finding the qualifying rows. An index is potentially useful if its first column is used in the search argument and the search argument establishes a lower bound, upper bound, or both to limit the search. In addition, if an index contains every column referenced in a query, even if none of those columns is the first column of the index, the index is considered useful.

An index that contains all the referenced columns is called a covering index and is one of the fastest ways to access data. The data pages do not have to be accessed at all, which can mean a substantial savings in physical I/O. For example, suppose that the Employees table has an index on last name, first name, and date of hire. The following query is covered by this index:

```
SELECT LastName, HireDate
FROM Employees
WHERE FirstName = 'Sven'
```

You might have more covering indexes than you're aware of. In SQL Server, a nonclustered index contains the clustered index key as part of the bookmark. So if the Employees table has a clustered index on the employee ID (EmployeeID) column, the nonclustered index described earlier, on LastName, FirstName, and HireDate, will also contain the EmployeeID for every data row, stored in the leaf level of the nonclustered index. Thus, the following is also a covered query:

```
SELECT EmployeeID, FirstName, LastName
FROM Employees
WHERE LastName LIKE 'B%'
```

Index Statistics

After the query optimizer finds a potentially useful index that matches the clause, it evaluates the index based on the selectivity of the clause. The optimizer checks the index's statistics. A histogram is created when the index is created on existing data, and the values are refreshed each time UPDATE STATISTICS runs. If the index is created before data exists in the table, no statistics are generated. The statistics will be misleading if they were generated when the dispersion of data values was significantly different from what appears in the current data in the table. However, SQL Server detects whether statistics are not up to date, and by default it automatically updates them during query optimization. We'll talk about when and how out-of-date statistics are detected in [Chapter 5](#).

SQL Server statistics contain a histogram consisting of a sampling of values for the index key (or the first column of the key for a composite index) based on the current data. To fully estimate the usefulness of an index, the optimizer also needs to know the number of pages in the table or index. This information is available through the compatibility view sysindexes or catalog view sys.allocation_units, which are discussed in *Inside SQL Server 2005: The Storage Engine*.

In addition to the histogram, the statistical information also includes details about the uniqueness (or density) of the data values encountered, which provides a measure of how selective the index is. The more selective an index is, the more useful it is, because higher selectivity means that more rows can be eliminated from consideration. A unique index, of course, is the most selective—by definition, each index entry can point to only one row. A unique index has a density value of 1/number of rows in the table.

Density values range from 0 through 1. Highly selective indexes have density values of 0.10 or lower. For example, a unique index on a table with 8,345 rows has a density of 0.00012 (1/8,345). If a nonunique nonclustered index has a density of 0.2165 on the same table, each index key can be expected to point to about 1,807 rows ($0.2165 \times 8,345$). This is probably not selective enough to be more efficient than just scanning the table, so this index most likely will not be considered useful. Because driving the query from a nonclustered index means that the pages must be retrieved in index order, an estimated 1,807 data page accesses (or logical reads) are needed if there is no clustered index on the table and the leaf level of the index contains the actual RID of the desired data row. The only time a data page doesn't need to be reaccessed is when the occasional coincidence occurs in which two adjacent index entries happen to point to the same data page.

Assume in this example that about 40 rows fit on a page, so there are about 209 total pages. The chance of two adjacent index entries pointing to the same page is only about 0.5 percent (1/209). The number of logical reads for just the data pages, not even counting the index I/O, is likely to be close to 1,807. If there is also a clustered index on the table, a couple of additional page accesses will be needed for each nonclustered key accessed. In contrast, the entire table can be scanned with just 209 logical reads—the number of pages in the table. In this case, the optimizer looks for better indexes or decides that a table scan is the best it can do.

The statistics histogram records steps (samples) for only the lead column of the index. This optimization takes into account that an index is useful only if its lead column is specified in a WHERE clause. However, density information is stored for multiple combinations of columns in a composite index. For example, if we have an index on (lastname, firstname), the statistics include a density value for lastname and another density value for the combination of lastname and firstname. This will become clearer when we look at some actual statistics information.

SQL Server defines density as follows:

$$\text{density} = 1/\text{cardinality of index keys}$$

The cardinality means how many unique values exist in the data. Statistics for a single column index consist of one histogram and one density value. The statistics for a composite index consist of one histogram for the first column in the index and density values for each prefix combination of columns (including the first column alone). The fact that density information is kept for all left-based subsets of columns helps the optimizer decide how useful the index is for joins.

Suppose, for example, that an index is composed of three key fields. The density on the first column might be 0.50, which is not too useful. But as you look at more key columns in the index, the number of rows pointed to is fewer than (or in the worst case, the same as) the first column, so the density value goes down. If you're looking at both the first and second columns, the density might be 0.25, which is somewhat better. And if you examine three columns, the density might be 0.03, which is highly selective. It doesn't make sense to refer to the density of only the second column. The lead column density is always needed.

The histogram contains up to 200 values of a given key column. In addition to the histogram, the statistics contain the following information:

- The time of the last statistics collection
- The number of rows used to produce the histogram and density information
- The average key length
- Densities for other combinations of columns

The following example uses the Orders table in the Northwind2 database. Assume that we have built a nonclustered composite index (cust_date_indx) on CustomerID and OrderDate.

```
CREATE INDEX cust_date_indx ON Orders(CustomerID, OrderDate)
```

The existing clustered index is on OrderID, so OrderID is considered part of the key in every nonclustered index row. We can use DBCC SHOW_STATISTICS to display the statistics information for this index:

```
DBCC SHOW_STATISTICS('Orders', 'cust_date_indx')
```

Here are the results:

Code View:

Statistics for INDEX 'cust_date_indx'.					
Updated	Rows	Rows Sampled	Steps	Density	Average key length
Jul 31 2007	830	830	89	0.0	22.0
All density Average Length Columns					
1.1235955E-2	10.0			CustomerID	
1.2150669E-3	18.0			CustomerID, OrderDate	
1.2048193E-3	22.0			CustomerID, OrderDate, OrderID	
RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS	
ALFKI	0.0	6.0	0	1	
ANATR	0.0	4.0	0	1	
ANTON	0.0	7.0	0	1	
AROUT	0.0	13.0	0	1	
BERGS	0.0	18.0	0	1	
BLAUS	0.0	7.0	0	1	
BLONP	0.0	11.0	0	1	
BOLID	0.0	3.0	0	1	
BONAP	0.0	17.0	0	1	
BOTTM	0.0	14.0	0	1	
BSBEV	0.0	10.0	0	1	
CACTU	0.0	6.0	0	1	
CENTC	0.0	1.0	0	1	
CHOPS	0.0	8.0	0	1	
COMM	0.0	5.0	0	1	
CONSH	0.0	3.0	0	1	
DRACD	0.0	6.0	0	1	
DUMON	0.0	4.0	0	1	
EASTC	0.0	8.0	0	1	
ERNSH	0.0	30.0	0	1	
FAMIA	0.0	7.0	0	1	
FOLIG	0.0	5.0	0	1	
FOLKO	0.0	19.0	0	1	
FRANK	0.0	15.0	0	1	
FRANR	0.0	3.0	0	1	
FRANS	0.0	6.0	0	1	
FURIB	0.0	8.0	0	1	
GALED	0.0	5.0	0	1	
GODOS	0.0	10.0	0	1	
GOURL	0.0	9.0	0	1	
GREAL	0.0	11.0	0	1	
GROSR	0.0	2.0	0	1	
HANAR	0.0	14.0	0	1	
HILAA	0.0	18.0	0	1	
HUNGC	0.0	5.0	0	1	
HUNGO	0.0	19.0	0	1	
ISLAT	0.0	10.0	0	1	
KOENE	0.0	14.0	0	1	
LACOR	0.0	4.0	0	1	
LAMAI	0.0	14.0	0	1	
LAUGB	0.0	3.0	0	1	
LAZYK	0.0	2.0	0	1	

LEHMS	0.0	15.0	0	1
LETSS	0.0	4.0	0	1
LILAS	0.0	14.0	0	1
LINOD	0.0	12.0	0	1
LONEP	0.0	8.0	0	1
MAGAA	0.0	10.0	0	1
MAISD	0.0	7.0	0	1
MEREP	0.0	13.0	0	1
MORGK	0.0	5.0	0	1
NORTS	0.0	3.0	0	1
OCEAN	0.0	5.0	0	1
OLDWO	0.0	10.0	0	1
OTTIK	0.0	10.0	0	1
PERIC	0.0	6.0	0	1
PICCO	0.0	10.0	0	1
PRINI	0.0	5.0	0	1
QUEDE	0.0	9.0	0	1
QUEEN	0.0	13.0	0	1
QUICK	0.0	28.0	0	1
RANCH	0.0	5.0	0	1
RATTC	0.0	18.0	0	1
REGGC	0.0	12.0	0	1
RICAR	0.0	11.0	0	1
RICSU	0.0	10.0	0	1
ROMEY	0.0	5.0	0	1
SANTG	0.0	6.0	0	1
SAVEA	0.0	31.0	0	1
SEVES	0.0	9.0	0	1
SIMOB	0.0	7.0	0	1
SPECD	0.0	4.0	0	1
SPLIR	0.0	9.0	0	1
SUPRD	0.0	12.0	0	1
THEBI	0.0	4.0	0	1
THECR	0.0	3.0	0	1
TOMSP	0.0	6.0	0	1
TORTU	0.0	10.0	0	1
TRADH	0.0	6.0	0	1
TRAIH	0.0	3.0	0	1
VAFFE	0.0	11.0	0	1
VICTE	0.0	10.0	0	1
VINET	0.0	5.0	0	1
WANDK	0.0	10.0	0	1
WARTH	0.0	15.0	0	1
WELLI	0.0	9.0	0	1
WHITC	0.0	14.0	0	1
WILMK	0.0	7.0	0	1
WOLZA	0.0	7.0	0	0.0

This output indicates that the statistics for this index were last updated on July 31, 2007. It also indicates that the table currently has 830 rows and the 89 steps indicates that 89 different rows were sampled. The average key length in the first row is for all columns of the index.

The next section shows the density values for each left-based subset of key columns. There are 89 different values for CustomerID, so the density is 1/89 or 1.1235955E-2 or 0.011235955. There are 823 different values for the combination of CustomerID and OrderDate, so the density is 1/823 or 1.2150669E-3 or 0.0012150669. When the third key column of OrderID (the clustered index key) is included, the key is now unique, so the density is 1 divided by the number of rows in the table, or 1/830, which is 1.2048193E-3 or 0.0012048193.

The statistics can store up to 200 sample values; the range of key values between each sample value is called a

step. The sample value is the endpoint of the range. The following values, or enough information to derive these values, are stored with each step.

RANGE_HI_KEY	A key value showing the upper boundary of the histogram step.
RANGE_ROWS	Specifies how many rows are inside the range (they are smaller than this RANGE_HI_KEY but bigger than the previous smaller RANGE_HI_KEY).
EQ_ROWS	Specifies how many rows are exactly equal to RANGE_HI_KEY.
AVG_RANGE_ROWS	Average number of rows per distinct value inside the range.
DISTINCT_RANGE_ROWS	Specifies how many distinct key values are inside this range (not including the previous key before RANGE_HI_KEY and RANGE_HI_KEY itself).

In the output shown previously, all the distinct key values in the first column of the index were one of the sample values, so there were no additional values in between the range endpoints. That's why so many of the values are 0 in the output.

In addition to statistics on indexes, SQL Server can also keep track of statistics on columns with no indexes. Knowing the density, or the likelihood of a particular value occurring, can help the optimizer determine an optimum processing strategy, even if SQL Server can't use an index to actually locate the values. We'll discuss column statistics in more detail when we look at statistics management later in this chapter.

Query optimization is probability based, which means that its conclusions can be wrong. The optimizer can make decisions that are not optimal even if they make sense from a probability standpoint. (As an analogy, you can think of national election polls, which are based on relatively small samples. The famous headline "Dewey Defeats Truman!" proves that such statistical sampling can be wrong.)

Index Cost

The second part of determining the selectivity of a SARG is calculating the estimated cost of the access methods that can be used. Even if a useful index is present, it might not be used if the optimizer determines that it is not the cheapest access method. One of the main components of the cost calculation, especially for queries involving only a single table, is the amount of logical I/O, which is the number of page accesses that are needed. Logical I/O is counted whether the pages are already in the memory cache or must be read from disk and brought into the cache. The term logical I/O is sometimes used to refer to I/O from cache, as if a read were either logical or physical, but this is a misnomer. Pages are always retrieved from the cache via a request to the buffer manager, so all reads are logical. If the pages are not already in the cache, they must be brought in first by the buffer manager. In those cases, the read is also physical. Only the buffer manager, which serves up the pages, knows whether a page is already in cache or must be accessed from disk and brought into cache. The ratio of the I/O already in the cache to the logical I/O is referred to as the cache-hit ratio, an important metric to watch.

The optimizer evaluates indexes to estimate the number of likely "hits" based on the density and step values in the statistics. Based on these values, it estimates how many rows qualify for the given SARG and how many logical reads would retrieve those qualifying rows. It might find multiple indexes that can find qualifying rows, or it might determine that just scanning the table and checking all the rows is best. It tries to find the access method that will require the fewest logical reads.

The access method, be it a clustered index scan or seek, seeks or scans one or more nonclustered indexes, a table scan, or another option, determines the estimated number of logical reads. This number can be very different for each method. Using an index to find qualifying rows is frequently referred to as an index driving the scan. When a nonclustered index drives the scan, the query optimizer assumes that the page containing

each identified qualifying row is probably not the same page accessed the last time. Because the pages must be retrieved according to the order of the index entries, retrieving the next row with a query driven by a nonclustered index will likely require that a different page from the one that contained the previous row be fetched because the two rows probably don't reside on the same page. They might reside on the same page by coincidence, but the optimizer correctly assumes that this will typically not be the case. (With 5,000 pages, the chance of this occurring is 1/5,000.) If the index is not clustered, data order is random with respect to the index key.

If the scan is driven from a clustered index, the next row is probably located on the same page as the previous row because the index leaf is, in fact, the data. The only time this is not the case when you use a clustered index is when the last row on a page has been retrieved; the next page must be accessed, and it will contain the next set of rows. But moving back and forth between the index and the data pages is not required.

There is, of course, a chance that a specific page might already be in cache when you use a nonclustered index. But a logical read will still occur even if the scan does not require physical I/O (a read from disk). Physical I/O is an order of magnitude more costly than I/O from cache. But don't assume that only the cost of physical I/O matters— reads from the cache are still far from free. To minimize logical I/O, the query optimizer tries to produce a plan that will result in the fewest number of page operations. In doing so, the optimizer will probably minimize physical I/O as well. For example, suppose we need to resolve a query with the following SARG:

```
WHERE Emp_Name BETWEEN 'Smith' AND 'Snow'
```

Here are the relevant index entries for a nonclustered index on Emp_Name:

Index_Key (Emp_Name)	Bookmarks: Found on Page(s)
Smith	1:267, 1:384, 1:512
Smyth	1:267
Snow	1:384, 1:512

If this table is a heap, when we use this nonclustered index, six logical reads are required to retrieve the data pages in addition to the I/O necessary to read the index. If none of the pages is already cached, the data access results in three physical I/O reads, assuming that the pages remain in the cache long enough to be reused for the subsequent rows (which is a good bet).

Suppose that all the index entries are on one leaf page and the index has one intermediate level. In this case, three I/O reads (logical and probably physical) are necessary to read the index (one root level, one intermediate level, and one leaf level). So chances are that driving this query via the nonclustered index requires about nine logical I/O reads, six of which are probably physical if the cache started out empty. The data pages are retrieved in the following order:

Page 1:267 to get row with Smith

Page 1:384 for Smith

Page 1:512 for Smith

Page 1:267 (again) for Smyth

Page 1:384 (again) for Snow

The number of logical reads would be more if the table also had a clustered index (for example, on the zip code field). Our leaf-level index entries might look like this:

Index_Key (Emp_Name)	Bookmarks: Clustered Key(s)
Smith	06403, 20191, 98370
Smyth	37027
Snow	22241, 80863

For each of the six nonclustered keys, we have to traverse the clustered index. You typically see about three to four times as many logical reads for traversing a nonclustered index for a table that also has a clustered index. However, if the indexes for this table are re-created so that a clustered index exists on the Emp_Name column, all six of the qualifying rows are probably located on the same page. The number of logical reads is probably only three (the index root, the intermediate index page, and the leaf index page, which is the data page), plus the final read that will retrieve all the qualifying rows. This scenario should make it clear to you why a clustered index can be so important to a range query.

With no clustered index, the optimizer has to choose between a plan that does a table scan and a plan that uses one or more nonclustered indexes. The number of logical I/O operations required for a scan of the table is equal to the number of pages in the table. A table scan starts at the first page and uses the Index Allocation Map (IAM) to access all the pages in the table. As the pages are read, the rows are evaluated to see whether they qualify based on the search criteria. Clearly, if the table from the previous example had fewer than nine total data pages, fewer logical reads would be needed to scan the whole table than to drive the scan off the nonclustered index, which was estimated to take nine logical reads.

The estimated logical reads for scanning qualifying rows is summarized in [Table 4-1](#). The access method with the least estimated cost is chosen based on this information.

Table 4-1. The Cost of Data Access for Tables with Different Index Structures

Access Method	Estimated Cost (in Logical Reads) ^[*]	Description
Table scan	The total number of data pages in the table.	
Clustered index	The number of levels in the index plus the number of data pages to scan. (Data pages to be scanned = number of qualifying rows/rows per data page.)	
Nonclustered index seek on a heap	The number of levels in the index plus the number of leaf pages plus the number of qualifying rows (a logical read for the data page of each qualifying row). The same data pages are often retrieved (from cache) many times, so the number of logical reads can be much higher than the number of pages in the table.	
Nonclustered index seek on a table with a clustered index	The number of levels in the index plus the number of leaf pages plus the number of qualifying rows times the cost of searching for a clustered index key.	
Covering nonclustered index	The number of levels in the index plus the number of leaf index pages (qualifying rows/rows per leaf page). The data page need not be accessed because all necessary information is in the index key.	

[*] All computations are rounded up to the next integer.

There is another type of index access that is combination of a nonclustered index scan and table lookup. Prior to SQL Server 2000, nonclustered index scans were only performed in the case of covering indexes, for which all the data a query needed could be found in the keys of a nonclustered index. In this case, SQL Server never needed to access the actual data pages at all. However, in SQL Server 2000 and SQL Server 2005 a new index access algorithm was added. Suppose you have a composite index on multiple columns and one of the nonleading columns is very selective. As an example, let's use the Person.Contacts table in the AdventureWorks database and build a nonclustered index on the Lastname and FirstName columns as shown:

```
USE AdventureWorks
CREATE INDEX NameIndex ON Person.Contact (LastName, FirstName)
```

A query that specifies only values for the FirstName column cannot use the index to directly seek to the appropriate rows, so a nonclustered index seek won't be chosen by the optimizer. However, if the table has more than a hundred pages or so, a scan might be too expensive. Another alternative is to scan the entire nonclustered leaf level, which contains all the LastName, FirstName combinations, and when SQL Server finds the FirstName value it is looking for, it can then look up (either by RID or clustered key depending on whether the table is a heap or not) to the underlying table. This solution is only beneficial if there are very few qualifying rows. For example, the following query will return 1,255 rows and the optimizer will choose a plan using a clustered index scan:

```
SELECT * FROM Person.Contact
WHERE FirstName like 'K%'
```

However, searching for a less common value, as in the following query, yields a different plan.

```
SELECT * FROM Person.Contact
WHERE FirstName like 'Y%'
```

This query returns only 37 rows, and if you examine the plan, you'll see that SQL Server will perform a scan on the NameIndex nonclustered index, and a key lookup into the clustered index for each of the 37 qualifying rows.

Remember that SQL Server only keeps a histogram for the leading column of an index. So how would the optimizer know that there were only a very few rows where the FirstName values starts with 'Y'? This is one of the places where column statistics can be useful. Statistics on the FirstName column will let the optimizer know approximately how many rows will satisfy the condition in the WHERE clause, even though there is no index to allow a direct seek to the specific FirstName values.

Using Multiple Indexes

The query optimizer can decide to use two or more nonclustered indexes to satisfy a single query. When more than one index is considered useful because of two or more SARGs, the cost estimate changes a bit from the formulas given in [Table 4-1](#). In addition to the I/O cost of finding the qualifying rows in each index, there is an additional cost of finding the intersection of the indexes so that SQL Server can determine which data rows satisfy all search conditions. Since each nonclustered index key contains a locator for the actual data, the results of the two index searches can be treated as worktables and joined together on the locator information.

Suppose we have the following query and we have separate nonclustered indexes on both the LastName and FirstName columns. In this case, we have a clustered index on EmployeeID, which is a unique value.

Code View:

```
USE Northwind2
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName BETWEEN 'Smith' AND 'Snow' AND FirstName BETWEEN 'Daniel' and 'David'
```

If the query optimizer decides to use both indexes, SQL Server will build two worktables with the results of each index search:

Index_Key (LastName)	Bookmark (Clustered Key)
Smith	66-712403
Smith	87-120191
Smith	76-137027
Smyth	11-983770
Snow	43-220191
Snow	21-780863

Index_Key (FirstName)	Bookmark (Clustered Key)
Daniel	11-983770
Daniel	77-321456
Danielle	66-712403
David	29-331218

These worktables are joined on the row locator information, which is unique for each row. Even if you don't declare a clustered index as unique, SQL Server adds a special uniqueifier where needed so there is always a unique row locator in every nonclustered index. You can get much more detailed information about the physical storage of your indexes in Inside SQL Server 2005: The Storage Engine. From the information above, we can see that only two rows have both their first name and last name values in the requested range. The result set is:

FirstName	LastName	EmployeeID
Daniel	Smythe	11-983770
Danielle	Smith	66-712403

In this example, the data in the two indexes is enough to satisfy the query. If we also want to see information such as address and phone number, the query optimizer uses the bookmark to find the actual data row by traversing the clustered index. It can also use multiple indexes if a single table has multiple SARGs that are OR'ed together. In this case, SQL Server finds the UNION of the rows returned by each index to determine all the qualifying rows. You can see the main differences if you take the same query and data and write an OR query instead of an AND:

Code View:

```
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName BETWEEN 'Smith' AND 'Snow' OR FirstName BETWEEN 'Daniel' AND 'David'
```

SQL Server can use the two indexes in the same way and build the same two worktables. However, it must UNION the two worktables together and remove duplicates. Once it removes the duplicates, it uses the bookmarks to find the rows in the table. All the result rows will have either a first name or a last name in the requested range, but not necessarily both. For example, we can return a row for Daniel Delaney or Bridgette Smith. If we don't remove duplicates, we end up with Daniel Smythe and Danielle Smith showing up twice because they meet both conditions of the SARGs.

Unusable Statistics

In two cases, the query optimizer can't use the statistics to estimate how many rows will satisfy a given SARG. First, if there are no statistics available, SQL Server obviously can't come up with any cost estimate. However, this situation is rare in SQL Server 2005 because by default every database has the two database options, AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS, set to ON. You can turn this behavior off at the index level or the database level, but this is usually not recommended. The second situation in which there are no usable statistics is when the values in a SARG are variables. Consider this example:

```
DECLARE @name varchar(30)
SET @name = 'Zelda'
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName > @name
```

Compilation and optimization takes place a batch at a time. When the SELECT query in preceding batch is optimized, the SET statement has not yet been executed and the value of @name is unknown. No specific value can be used to compare the steps in the index's statistics information, so the query optimizer must guess what the value will be.

Note



Don't confuse variables with parameters even though their syntax is nearly identical. A variable's value is never known until the statement is actually executed; it is never known at compile time. Stored procedure parameters are known when the procedure is compiled because compilation and optimization don't even take place until the procedure is actually called with specific values for the parameters.

If the query optimizer can't use the statistics for either of the reasons mentioned, the server uses fixed percentages to estimate how many rows will satisfy a given SARG. These percentages, shown here, depend on the comparison operator used. These fixed percentages can be grossly inaccurate for your specific data, however, so make sure that your statistics are up to date and usable whenever possible.

Operator	Percentage of Rows
=	10
>	30
<	30
BETWEEN	9

When the SARG involves an equality, the situation is usually handled in a special way. The 10 percent estimate shown above is actually used only in the unusual case in which there are no statistics at all. If there are statistics, they can be used. Even if we don't have a particular value to compare against the steps in the statistics histogram, the density information can be used. That information basically tells the query optimizer the expected number of duplicates, so when the SARG involves looking for one specific value, it assumes that the value occurs the average number of times.

An even more special case occurs when the query optimizer recognizes an equality in the WHERE clause and the index is unique. Because this combination yields an exact match and always returns at most one row, the

query optimizer doesn't have to use statistics. For queries of one row that use an exact match such as a lookup by a primary key, a unique nonclustered index is highly efficient. In fact, many environments probably shouldn't "waste" their clustered index on the primary key. If access via the complete primary key is common, it might be beneficial to specify NONCLUSTERED when you declare the primary key and save the clustered index for another type of access (such as a range query) that can benefit more from the clustered index.

Chapter 4. Troubleshooting Query Performance

by Kalen Delaney and Craig Freedman

In this chapter:	
Compilation and Optimization	199
Detecting Problems in Plans	218
Monitoring Query Performance	221
Query Improvements	225
Query Processing Best Practices	273
Summary	275

As mentioned in [Chapter 3](#), "Query Execution," the SQL Server query processor consists of two components: the query optimizer and the query execution engine. In [Chapter 3](#), we looked at details of many different types of query plans and discussed the process of query execution. In the first part of this chapter, we'll talk about the query optimizer, which is responsible for generating good query plans. The optimizer includes subcomponents to gather statistics, perform cardinality estimation using these statistics, estimate the cost of query plans, and, of course, explore alternative query plans in order to generate a good plan for execution. In the second part of the chapter, we'll discuss techniques to detect suboptimal query plans. The third section describes techniques for improving your query's performance and includes a discussion of the new query optimization hints introduced in SQL Server 2005. The chapter ends with a list of best-practice guidelines to keep in mind when building any SQL Server-based application.

Compilation and Optimization

It's important to be aware that the compilation and optimization process is completely separate from the process of execution. The gap between when SQL Server compiles a query and when the query is executed can be as short as a few microseconds or as long as several days. Therefore, during compilation (which includes optimization), SQL Server has to determine what kind of available information will be useful when the query is executed. Not everything that is true at compilation time will be true at execution time. For example, if the optimizer took into account how many concurrent users were trying to access the same data as the query being optimized, that information could change dramatically by the time the query was executed. Even the amount of data and the data distribution can change between compilation and execution, so we really need to consider the compilation phase of query processing completely separately from the execution phase.

Compilation

When a query is ready to be processed by SQL Server, an execution plan for the query might already be available in SQL Server's plan cache. If no usable plan is available, the query must be compiled. The methods

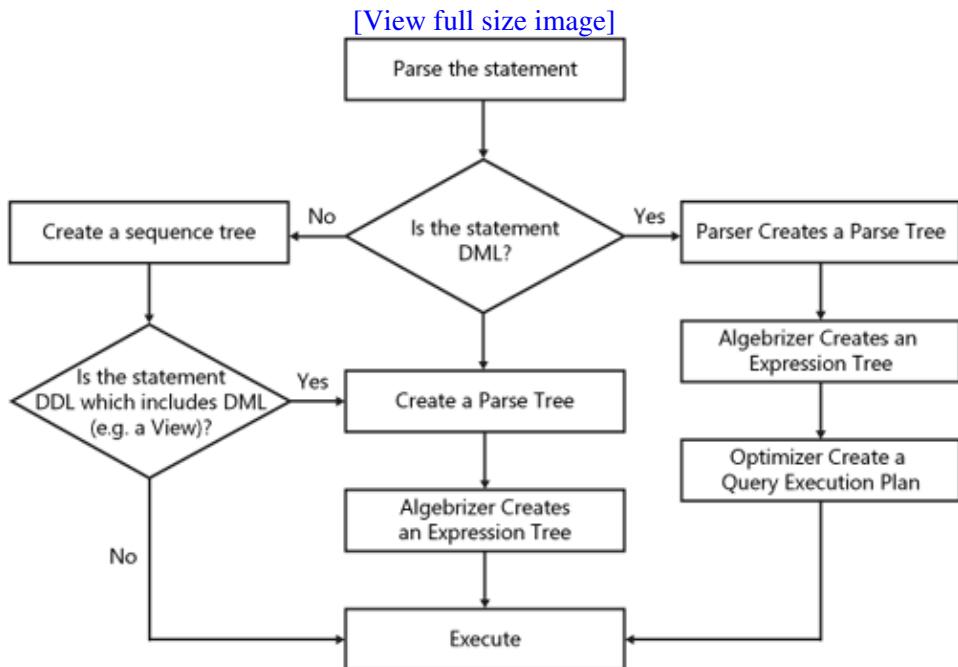
by which execution plans can be saved and reused are discussed in detail in [Chapter 5](#), "Plan Caching and Recompilation." In this section, we're assuming there is no available plan and your query must be compiled. The compilation process encompasses a few steps. First, your query is processed by the parser and then by a component called the algebrizer. The parser handles the process of dissecting and transforming your SQL statements into compiler-ready data structures. Parsing also includes validating the syntax to ensure that it is legal. If the parser determines that your SQL statement is a DML statement, it creates an expression parse tree. For non-DML statements, such as DDL for object creation statements, the parser produces a sequence tree. The sequence tree is one of the few data structures in SQL Server 2005 that has survived from SQL Server 6.5. The main difference between a sequence tree and a parse tree is that the sequence tree is a binary tree, whereas the parse tree can have multiple children attached to each node. In addition, some DDL statements need to include a parse tree as well as a sequence tree. For example, in a CREATE VIEW statement, the SELECT statement comprising the view definition is parsed as an expression tree while the wrapper is parsed as a sequence tree. Also, if a CREATE TABLE statement includes default values or check constraints, those get parsed as expression trees.

Parsing doesn't include things such as checking for valid table and column names, which are dealt with by the algebrizer. The algebrizer basically determines the characteristics of the objects that you reference inside your SQL statements and checks whether the semantics you're asking for make sense. For example, it is semantically illogical to try to execute a table. In addition, the algebrizer makes sure your objects are valid, including verification that the column names referenced actually do exist in the referenced tables and binding the object IDs and column IDs to the object references. The algebrizer only acts on expression parse trees (not on sequence trees) and its output is an algebrized expression tree.

The algebrized expression tree is then compiled by the query optimizer. After the execution plan is generated, it is placed in cache and then executed.

[Figure 4-1](#) shows the flow of this compilation process, leading up to query execution.

Figure 4-1. The steps in compiling a statement



Optimization

SQL Server's query optimizer is a cost-based optimizer, which means that it tries to come up with a low-cost execution plan for each SQL statement. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The query optimizer must analyze possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it tries to find an execution plan that has a cost reasonably close to the theoretical minimum. Later in this section, we'll discuss some of the ways that the optimizer can reduce the amount of time it needs to spend on optimization.

The lowest estimated cost is not simply the lowest resource cost; the query optimizer chooses a plan that balances the amount of time needed to return results to the user with a reasonable cost in resources. For example, processing a query in parallel (using multiple CPUs simultaneously for the same query) typically uses more resources than processing it serially using a single CPU, but the query completes much faster. The optimizer will choose a parallel execution plan to return results if the load on the server will not be adversely affected.

Optimization itself involves many steps. The first step is called trivial plan optimization. The idea behind trivial plan optimization is that cost-based optimization is expensive to run. The optimizer can try many possible variations in looking for a cheap plan. If SQL Server knows that there is only one really viable plan for a query, it can avoid a lot of work. A common example is a query that consists of an INSERT with a VALUES clause. There is only one possible plan. Another example is a SELECT statement, in which the columns in the WHERE clause include all keys of a unique index and that index is the only one that is relevant; that is, no other index has that set of columns in it. In these two cases, SQL Server should just generate the plan and not try to find something better. Trivial plan optimization finds the really obvious plans that are typically very inexpensive. This saves the optimizer from having to consider every possible plan, which can be costly and can outweigh any benefit provided by well-optimized queries. You can determine that the trivial plan optimization has found a plan in one of two ways. You can look at the graphical execution plan, and open up the properties window. Clicking on the left-most iterator will show properties for the entire query. In the left column of the properties list, look for Optimization Level. The right column will show either TRIVIAL or FULL. Alternatively, you can generate an XML plan for a query. Near the top of the XML document, within an element called StmtSimple, there will be an attribute called StatementOptmLevel. A value of TRIVIAL for this attribute indicates a plan found by the trivial plan optimization.

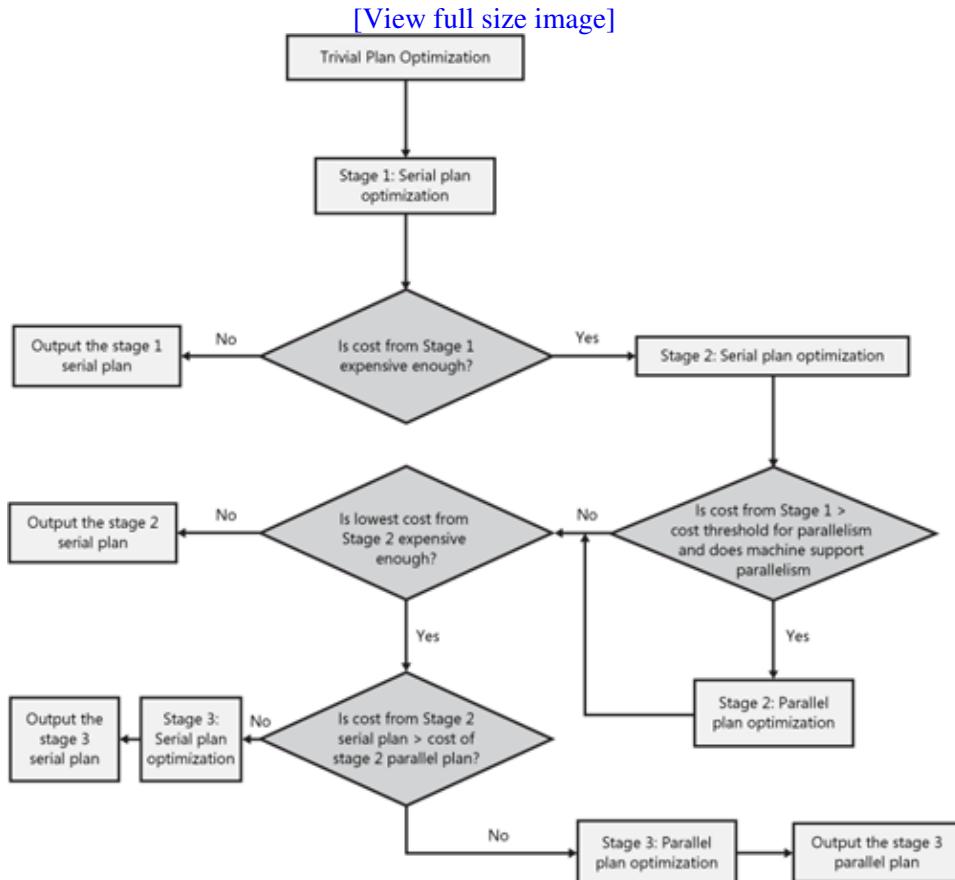
If the trivial plan optimization doesn't find a simple plan, SQL Server can perform some simplifications, which are usually syntactic transformations of the query itself, to look for commutative properties and operations that can be rearranged. SQL Server can perform operations that don't require looking at the cost or analyzing what indexes are available but that can result in a more efficient query. SQL Server then loads up the metadata, including the statistical information on the indexes, and then the optimizer goes through a series of phases of cost-based optimization.

Cost-based optimization is designed as a set of transformation rules that try various permutations of indexes, join strategies, aggregation strategies, table orderings, etc. Because of the number of potential plans in SQL Server 2005, if the optimizer just ran through all the combinations and produced a plan, the optimization process would take a long time. So optimization is broken up into three stages. Each stage is a set of rules. After each stage, SQL Server evaluates the cost of any resulting plan, and if the plan is cheap enough, that plan is considered the final plan. If the plan is not cheap enough, the optimizer runs the next stage, which extends the set of rules.

Stage 1 considers only serial plans (single processor) and if it finds a good enough serial plan, the optimizer does not proceed to the subsequent stages. If the cost from stage 1 is not cheap enough, stage 2 first tries to find a serial plan, and then considers a parallel plan. The parallel plan will only be considered if the machine supports parallelism and is configured to use multiple processors for a single query, and if the configuration option cost threshold for parallelism is less than the cost at the end of stage 1.

If the best cost after stage 2 is still not cheap enough, the optimizer proceeds to stage 3. If the serial plan cost at the end of stage 2 is greater than the parallel plan cost at the end of stage 2, stage 3 will only consider parallel plans. Otherwise the optimizer will only consider serial plans. At the end of stage 3, the cheapest plan is considered the final plan and that is the plan that will be executed. [Figure 4-2](#) shows the flow of processing through the optimizer.

Figure 4-2. The steps in optimizing a statement



How the Query Optimizer Works

Monitoring and tuning queries are essential steps in optimizing performance. Knowing how the query optimizer works can be helpful as you think about how to write a good query or what indexes to define. However, you should guard against outsmarting yourself and trying to predict what the optimizer will do. You might miss some good options this way. Try to write your queries in the most intuitive way you can, and then try to tune them only if their performance doesn't seem good enough.

Nevertheless, insight into how the optimizer works is certainly useful. It can help you understand why certain indexes are recommended over others, and it can help you understand the query plan output in SQL Server Management Studio. For each table involved in the query, the query optimizer evaluates the search conditions in the WHERE clause and considers which indexes are available to narrow the scan of a table. That is, the optimizer evaluates to what extent an index can exclude rows from consideration. The more rows that can be excluded, the better, because that leaves fewer rows to process.

As discussed in [Chapter 3](#), SQL Server can process joins, aggregations, unions, etc. in many different ways.

For each potential strategy, the optimizer determines which index or indexes might be useful and then estimates a cost, taking into account the number of logical reads and the memory resources that are required and available. The optimizer compares the cost of each potential plan and chooses the plan with the lowest estimate. Prior to determining the useful indexes, the optimizer performs a step that we can call Query Analysis, which we'll look at next. After that, we'll examine how SQL Server determines the usefulness of your existing indexes.

Query Analysis

During the query analysis phase, the query optimizer looks at each clause of the query and determines whether it can be useful in limiting how much data must be scanned— that is, whether the clause is useful as a search argument (SARG) or as part of the join criteria. A clause that can be used as a SARG is referred to as sargable, or optimizable, and can make use of an index for faster retrieval.

A SARG limits a search because it specifies an exact match, a range of values, or a conjunction of two or more limiting conditions combined by AND. A SARG contains a constant expression (or a variable that is resolved to a constant) that acts on a column by using an operator. It has the form

```
column inclusive_operator <constant or variable>
```

or

```
<constant or variable> inclusive_operator column
```

The column name can appear on one side of the operator, and the constant or variable can appear on the other side. If a column appears on both sides of the operator, the clause is not sargable. Sargable operators include =, >, <, =>, <=, BETWEEN, and sometimes LIKE. Whether LIKE is sargable depends on the type and position of wildcards used. For example, LIKE 'Jon%' is sargable but LIKE '%Jon' is not because the wildcard (%) at the beginning prevents the use of an index. Here are some SARG examples:

```
name = 'jones'
salary > 40000
60000 < salary
department = 'sales'
name = 'jones' AND salary > 100000
name LIKE 'dail%'
```

A single SARG can include many conditions if they are AND'ed together. That is, one index might be able to operate on all the conditions that are AND'ed together. In the example above, there might be an index on (name, salary), so the entire clause name = 'jones' AND salary > 100000 can be considered one SARG and can be evaluated for qualifying rows using one index. If OR is used instead of AND in this example, a single index scan cannot be used to qualify both terms. The reason should be clear—if the lead field in the index key is name, the index is useful for finding just the 'jones' rows. If the condition in the WHERE clause includes AND salary > 100000, the second field of the index also qualifies those rows. That is, all the rows that have a name value of 'jones' can be further restricted to find the rows with 'jones' and an appropriate salary. The set of rows that meets both of the AND conditions is a subset of the rows that meet just the first condition on name. But if the second criterion is OR salary > 100000, the index is not useful in the same way because all the rows would need to be examined, not just the 'jones' entries. The set of rows that meet either one of the conditions is a superset of the rows that meet just the first condition on name.

A phone book analogy can also be useful. If you want to find people who have the surname "Jones" AND live on 5th Avenue, using the phone book can help greatly reduce the size of your search. But if you want to find people who have the name "Jones" OR live on 5th Avenue, you have to scan every entry in the entire phone book. This assumes that you have only one phone book that is sorted alphabetically by name. If you have two phone books, one sorted by name and one sorted by street, that's another story, which we'll discuss a bit later.

An expression that is not sargable cannot limit the search. That is, SQL Server must evaluate every row to determine whether it meets the conditions in the WHERE clause. So an index is not useful to nonsargable expressions. Typical nonsargable expressions include referencing a column in an expression or function, certain negation operators such as NOT, NOT IN, and NOT LIKE and strings with a wildcard at the beginning. Don't extrapolate too far and think that this means using a nonsargable clause always results in a table scan. An index is not useful to the nonsargable clause, but there might be indexes useful to other SARGs in the query. Queries often have multiple clauses, so a great index for one or more of the other clauses might be available. Here are some examples of nonsargable clauses:

```
ABS(price) < 4
name LIKE '%jon%'
name = 'jones' OR salary > 100000
```

Note



The last example above is not a single search argument, but each expression on either side of the OR is individually sargable. So a single index won't be used to evaluate both expressions, as it might if the operator is AND. A separate index can still be useful to each expression.

Let's look at a specific example. The Northwind2 database includes a table called Order Details, which has a nonclustered index on the column ProductID. When a query contains a SARG involving ProductID, the query optimizer has to determine whether using the nonclustered index will be the fastest way to access the qualifying rows. Remember that a nonclustered index contains every key value in its leaf level, with a bookmark pointing to the actual location of the row having that key. If too many rows satisfy the SARG's condition, the optimizer might decide it's too expensive to use the index and opt instead to scan the entire table. In addition to metadata keeping track of the key columns and properties of each index, SQL Server also maintains statistical information about the distribution of all the different key values in each index. We'll look at statistics in detail a bit later, but for now, be aware that examining the statistics is part of determining whether an index is useful.

The first query below has a SARG involving the CustomerID column in the BigOrders table in the Northwind2 database. SQL Server is able to use the statistical information to estimate that 25 rows will meet the condition in the WHERE clause. The optimizer will choose to use the index on CustomerID.

```
USE Northwind2
SELECT * FROM BigOrders
WHERE CustomerID = 'Ocean';
```

In the next query, because of the presence of the function UPPER, there is no SARG and the index on CustomerID is not even considered. The optimizer has no real choice other than to scan the entire table using a Clustered Index Scan.

```
USE Northwind2
```

```
SELECT * FROM BigOrders
WHERE UPPER(CustomerID) = 'OCEAN';
```

Index Selection

During the second phase of query optimization, index selection, the query optimizer determines whether an index exists for a sargable clause, assesses the index's usefulness by determining the selectivity of the clause (that is, how many rows will be returned), and estimates the cost of finding the qualifying rows. An index is potentially useful if its first column is used in the search argument and the search argument establishes a lower bound, upper bound, or both to limit the search. In addition, if an index contains every column referenced in a query, even if none of those columns is the first column of the index, the index is considered useful.

An index that contains all the referenced columns is called a covering index and is one of the fastest ways to access data. The data pages do not have to be accessed at all, which can mean a substantial savings in physical I/O. For example, suppose that the Employees table has an index on last name, first name, and date of hire. The following query is covered by this index:

```
SELECT LastName, HireDate
FROM Employees
WHERE FirstName = 'Sven'
```

You might have more covering indexes than you're aware of. In SQL Server, a nonclustered index contains the clustered index key as part of the bookmark. So if the Employees table has a clustered index on the employee ID (EmployeeID) column, the nonclustered index described earlier, on LastName, FirstName, and HireDate, will also contain the EmployeeID for every data row, stored in the leaf level of the nonclustered index. Thus, the following is also a covered query:

```
SELECT EmployeeID, FirstName, LastName
FROM Employees
WHERE LastName LIKE 'B%'
```

Index Statistics

After the query optimizer finds a potentially useful index that matches the clause, it evaluates the index based on the selectivity of the clause. The optimizer checks the index's statistics. A histogram is created when the index is created on existing data, and the values are refreshed each time UPDATE STATISTICS runs. If the index is created before data exists in the table, no statistics are generated. The statistics will be misleading if they were generated when the dispersion of data values was significantly different from what appears in the current data in the table. However, SQL Server detects whether statistics are not up to date, and by default it automatically updates them during query optimization. We'll talk about when and how out-of-date statistics are detected in [Chapter 5](#).

SQL Server statistics contain a histogram consisting of a sampling of values for the index key (or the first column of the key for a composite index) based on the current data. To fully estimate the usefulness of an index, the optimizer also needs to know the number of pages in the table or index. This information is available through the compatibility view sysindexes or catalog view sys.allocation_units, which are discussed in Inside SQL Server 2005: The Storage Engine.

In addition to the histogram, the statistical information also includes details about the uniqueness (or density) of the data values encountered, which provides a measure of how selective the index is. The more selective an index is, the more useful it is, because higher selectivity means that more rows can be eliminated from consideration. A unique index, of course, is the most selective—by definition, each index entry can point to only one row. A unique index has a density value of 1/number of rows in the table.

Density values range from 0 through 1. Highly selective indexes have density values of 0.10 or lower. For example, a unique index on a table with 8,345 rows has a density of 0.00012 (1/8,345). If a nonunique nonclustered index has a density of 0.2165 on the same table, each index key can be expected to point to about 1,807 rows ($0.2165 \times 8,345$). This is probably not selective enough to be more efficient than just scanning the table, so this index most likely will not be considered useful. Because driving the query from a nonclustered index means that the pages must be retrieved in index order, an estimated 1,807 data page accesses (or logical reads) are needed if there is no clustered index on the table and the leaf level of the index contains the actual RID of the desired data row. The only time a data page doesn't need to be reaccessed is when the occasional coincidence occurs in which two adjacent index entries happen to point to the same data page.

Assume in this example that about 40 rows fit on a page, so there are about 209 total pages. The chance of two adjacent index entries pointing to the same page is only about 0.5 percent (1/209). The number of logical reads for just the data pages, not even counting the index I/O, is likely to be close to 1,807. If there is also a clustered index on the table, a couple of additional page accesses will be needed for each nonclustered key accessed. In contrast, the entire table can be scanned with just 209 logical reads—the number of pages in the table. In this case, the optimizer looks for better indexes or decides that a table scan is the best it can do.

The statistics histogram records steps (samples) for only the lead column of the index. This optimization takes into account that an index is useful only if its lead column is specified in a WHERE clause. However, density information is stored for multiple combinations of columns in a composite index. For example, if we have an index on (lastname, firstname), the statistics include a density value for lastname and another density value for the combination of lastname and firstname. This will become clearer when we look at some actual statistics information.

SQL Server defines density as follows:

$\text{density} = 1/\text{cardinality of index keys}$

The cardinality means how many unique values exist in the data. Statistics for a single column index consist of one histogram and one density value. The statistics for a composite index consist of one histogram for the first column in the index and density values for each prefix combination of columns (including the first column alone). The fact that density information is kept for all left-based subsets of columns helps the optimizer decide how useful the index is for joins.

Suppose, for example, that an index is composed of three key fields. The density on the first column might be 0.50, which is not too useful. But as you look at more key columns in the index, the number of rows pointed to is fewer than (or in the worst case, the same as) the first column, so the density value goes down. If you're looking at both the first and second columns, the density might be 0.25, which is somewhat better. And if you examine three columns, the density might be 0.03, which is highly selective. It doesn't make sense to refer to the density of only the second column. The lead column density is always needed.

The histogram contains up to 200 values of a given key column. In addition to the histogram, the statistics contain the following information:

- The time of the last statistics collection
- The number of rows used to produce the histogram and density information
- The average key length
- Densities for other combinations of columns

The following example uses the Orders table in the Northwind2 database. Assume that we have built a nonclustered composite index (cust_date_indx) on CustomerID and OrderDate.

```
CREATE INDEX cust_date_indx ON Orders(CustomerID, OrderDate)
```

The existing clustered index is on OrderID, so OrderID is considered part of the key in every nonclustered index row. We can use DBCC SHOW_STATISTICS to display the statistics information for this index:

```
DBCC SHOW_STATISTICS('Orders', 'cust_date_indx')
```

Here are the results:

Code View:

Statistics for INDEX 'cust_date_indx'.					
Updated	Rows	Rows Sampled	Steps	Density	Average key length
Jul 31 2007	830	830	89	0.0	22.0
<hr/>					
All density	Average Length	Columns	<hr/>		
1.1235955E-2	10.0	CustomerID	<hr/>		
1.2150669E-3	18.0	CustomerID, OrderDate	<hr/>		
1.2048193E-3	22.0	CustomerID, OrderDate, OrderID	<hr/>		
RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS	<hr/>
ALFKI	0.0	6.0	0	1	<hr/>
ANATR	0.0	4.0	0	1	<hr/>
ANTON	0.0	7.0	0	1	<hr/>
AROUT	0.0	13.0	0	1	<hr/>
BERGS	0.0	18.0	0	1	<hr/>
BLAUS	0.0	7.0	0	1	<hr/>
BLONP	0.0	11.0	0	1	<hr/>
BOLID	0.0	3.0	0	1	<hr/>
BONAP	0.0	17.0	0	1	<hr/>
BOTTM	0.0	14.0	0	1	<hr/>
BSBEV	0.0	10.0	0	1	<hr/>
CACTU	0.0	6.0	0	1	<hr/>
CENTC	0.0	1.0	0	1	<hr/>
CHOPS	0.0	8.0	0	1	<hr/>
COMMI	0.0	5.0	0	1	<hr/>
CONSH	0.0	3.0	0	1	<hr/>
DRACD	0.0	6.0	0	1	<hr/>
DUMON	0.0	4.0	0	1	<hr/>
EASTC	0.0	8.0	0	1	<hr/>
ERNSH	0.0	30.0	0	1	<hr/>
FAMIA	0.0	7.0	0	1	<hr/>
FOLIG	0.0	5.0	0	1	<hr/>
FOLKO	0.0	19.0	0	1	<hr/>
FRANK	0.0	15.0	0	1	<hr/>
FRANR	0.0	3.0	0	1	<hr/>
FRANS	0.0	6.0	0	1	<hr/>
FURIB	0.0	8.0	0	1	<hr/>
GALED	0.0	5.0	0	1	<hr/>
GODOS	0.0	10.0	0	1	<hr/>
GOURL	0.0	9.0	0	1	<hr/>
GREAL	0.0	11.0	0	1	<hr/>
GROSR	0.0	2.0	0	1	<hr/>
HANAR	0.0	14.0	0	1	<hr/>
HILAA	0.0	18.0	0	1	<hr/>

HUNG	0.0	5.0	0	1
HUNGO	0.0	19.0	0	1
ISLAT	0.0	10.0	0	1
KOENE	0.0	14.0	0	1
LACOR	0.0	4.0	0	1
LAMAI	0.0	14.0	0	1
LAUGB	0.0	3.0	0	1
LAZYK	0.0	2.0	0	1
LEHMS	0.0	15.0	0	1
LETSS	0.0	4.0	0	1
LILAS	0.0	14.0	0	1
LINOD	0.0	12.0	0	1
LONEP	0.0	8.0	0	1
MAGAA	0.0	10.0	0	1
MAISD	0.0	7.0	0	1
MEREP	0.0	13.0	0	1
MORGK	0.0	5.0	0	1
NORTS	0.0	3.0	0	1
OCEAN	0.0	5.0	0	1
OLDWO	0.0	10.0	0	1
OTTIK	0.0	10.0	0	1
PERIC	0.0	6.0	0	1
PICCO	0.0	10.0	0	1
PRINI	0.0	5.0	0	1
QUEDE	0.0	9.0	0	1
QUEEN	0.0	13.0	0	1
QUICK	0.0	28.0	0	1
RANCH	0.0	5.0	0	1
RATTC	0.0	18.0	0	1
REGGC	0.0	12.0	0	1
RICAR	0.0	11.0	0	1
RICSU	0.0	10.0	0	1
ROMEY	0.0	5.0	0	1
SANTG	0.0	6.0	0	1
SAVEA	0.0	31.0	0	1
SEVES	0.0	9.0	0	1
SIMOB	0.0	7.0	0	1
SPECD	0.0	4.0	0	1
SPLIR	0.0	9.0	0	1
SUPRD	0.0	12.0	0	1
THEBI	0.0	4.0	0	1
THECR	0.0	3.0	0	1
TOMSP	0.0	6.0	0	1
TORTU	0.0	10.0	0	1
TRADH	0.0	6.0	0	1
TRAIH	0.0	3.0	0	1
VAFFE	0.0	11.0	0	1
VICTE	0.0	10.0	0	1
VINET	0.0	5.0	0	1
WANDK	0.0	10.0	0	1
WARTH	0.0	15.0	0	1
WELLI	0.0	9.0	0	1
WHITC	0.0	14.0	0	1
WILMK	0.0	7.0	0	1
WOLZA	0.0	7.0	0	0.0

This output indicates that the statistics for this index were last updated on July 31, 2007. It also indicates that the table currently has 830 rows and the 89 steps indicates that 89 different rows were sampled. The average key length in the first row is for all columns of the index.

The next section shows the density values for each left-based subset of key columns. There are 89 different values for CustomerID, so the density is 1/89 or 1.1235955E-2 or 0.011235955. There are 823 different values for the combination of CustomerID and OrderDate, so the density is 1/823 or 1.2150669E-3 or 0.0012150669. When the third key column of OrderID (the clustered index key) is included, the key is now unique, so the density is 1 divided by the number of rows in the table, or 1/830, which is 1.2048193E-3 or 0.0012048193.

The statistics can store up to 200 sample values; the range of key values between each sample value is called a step. The sample value is the endpoint of the range. The following values, or enough information to derive these values, are stored with each step.

RANGE_HI_KEY	A key value showing the upper boundary of the histogram step.
RANGE_ROWS	Specifies how many rows are inside the range (they are smaller than this RANGE_HI_KEY but bigger than the previous smaller RANGE_HI_KEY).
EQ_ROWS	Specifies how many rows are exactly equal to RANGE_HI_KEY.
AVG_RANGE_ROWS	Average number of rows per distinct value inside the range.
DISTINCT_RANGE_ROWS	Specifies how many distinct key values are inside this range (not including the previous key before RANGE_HI_KEY and RANGE_HI_KEY itself).

In the output shown previously, all the distinct key values in the first column of the index were one of the sample values, so there were no additional values in between the range endpoints. That's why so many of the values are 0 in the output.

In addition to statistics on indexes, SQL Server can also keep track of statistics on columns with no indexes. Knowing the density, or the likelihood of a particular value occurring, can help the optimizer determine an optimum processing strategy, even if SQL Server can't use an index to actually locate the values. We'll discuss column statistics in more detail when we look at statistics management later in this chapter.

Query optimization is probability based, which means that its conclusions can be wrong. The optimizer can make decisions that are not optimal even if they make sense from a probability standpoint. (As an analogy, you can think of national election polls, which are based on relatively small samples. The famous headline "Dewey Defeats Truman!" proves that such statistical sampling can be wrong.)

Index Cost

The second part of determining the selectivity of a SARG is calculating the estimated cost of the access methods that can be used. Even if a useful index is present, it might not be used if the optimizer determines that it is not the cheapest access method. One of the main components of the cost calculation, especially for queries involving only a single table, is the amount of logical I/O, which is the number of page accesses that are needed. Logical I/O is counted whether the pages are already in the memory cache or must be read from disk and brought into the cache. The term logical I/O is sometimes used to refer to I/O from cache, as if a read were either logical or physical, but this is a misnomer. Pages are always retrieved from the cache via a request to the buffer manager, so all reads are logical. If the pages are not already in the cache, they must be brought in first by the buffer manager. In those cases, the read is also physical. Only the buffer manager, which serves up the pages, knows whether a page is already in cache or must be accessed from disk and brought into cache. The ratio of the I/O already in the cache to the logical I/O is referred to as the cache-hit ratio, an important metric to watch.

The optimizer evaluates indexes to estimate the number of likely "hits" based on the density and step values in the statistics. Based on these values, it estimates how many rows qualify for the given SARG and how many logical reads would retrieve those qualifying rows. It might find multiple indexes that can find qualifying

rows, or it might determine that just scanning the table and checking all the rows is best. It tries to find the access method that will require the fewest logical reads.

The access method, be it a clustered index scan or seek, seeks or scans one or more nonclustered indexes, a table scan, or another option, determines the estimated number of logical reads. This number can be very different for each method. Using an index to find qualifying rows is frequently referred to as an index driving the scan. When a nonclustered index drives the scan, the query optimizer assumes that the page containing each identified qualifying row is probably not the same page accessed the last time. Because the pages must be retrieved according to the order of the index entries, retrieving the next row with a query driven by a nonclustered index will likely require that a different page from the one that contained the previous row be fetched because the two rows probably don't reside on the same page. They might reside on the same page by coincidence, but the optimizer correctly assumes that this will typically not be the case. (With 5,000 pages, the chance of this occurring is 1/5,000.) If the index is not clustered, data order is random with respect to the index key.

If the scan is driven from a clustered index, the next row is probably located on the same page as the previous row because the index leaf is, in fact, the data. The only time this is not the case when you use a clustered index is when the last row on a page has been retrieved; the next page must be accessed, and it will contain the next set of rows. But moving back and forth between the index and the data pages is not required.

There is, of course, a chance that a specific page might already be in cache when you use a nonclustered index. But a logical read will still occur even if the scan does not require physical I/O (a read from disk). Physical I/O is an order of magnitude more costly than I/O from cache. But don't assume that only the cost of physical I/O matters—reads from the cache are still far from free. To minimize logical I/O, the query optimizer tries to produce a plan that will result in the fewest number of page operations. In doing so, the optimizer will probably minimize physical I/O as well. For example, suppose we need to resolve a query with the following SARG:

```
WHERE Emp_Name BETWEEN 'Smith' AND 'Snow'
```

Here are the relevant index entries for a nonclustered index on Emp_Name:

Index_Key (Emp_Name)	Bookmarks: Found on Page(s)
Smith	1:267, 1:384, 1:512
Smyth	1:267
Snow	1:384, 1:512

If this table is a heap, when we use this nonclustered index, six logical reads are required to retrieve the data pages in addition to the I/O necessary to read the index. If none of the pages is already cached, the data access results in three physical I/O reads, assuming that the pages remain in the cache long enough to be reused for the subsequent rows (which is a good bet).

Suppose that all the index entries are on one leaf page and the index has one intermediate level. In this case, three I/O reads (logical and probably physical) are necessary to read the index (one root level, one intermediate level, and one leaf level). So chances are that driving this query via the nonclustered index requires about nine logical I/O reads, six of which are probably physical if the cache started out empty. The data pages are retrieved in the following order:

Page 1:267 to get row with Smith

Page 1:384 for Smith

Page 1:512 for Smith

Page 1:267 (again) for Smyth

Page 1:384 (again) for Snow

Page 1:512 (again) for Snow

The number of logical reads would be more if the table also had a clustered index (for example, on the zip code field). Our leaf-level index entries might look like this:

Index_Key (Emp_Name)	Bookmarks: Clustered Key(s)
Smith	06403, 20191, 98370
Smyth	37027
Snow	22241, 80863

For each of the six nonclustered keys, we have to traverse the clustered index. You typically see about three to four times as many logical reads for traversing a nonclustered index for a table that also has a clustered index. However, if the indexes for this table are re-created so that a clustered index exists on the Emp_Name column, all six of the qualifying rows are probably located on the same page. The number of logical reads is probably only three (the index root, the intermediate index page, and the leaf index page, which is the data page), plus the final read that will retrieve all the qualifying rows. This scenario should make it clear to you why a clustered index can be so important to a range query.

With no clustered index, the optimizer has to choose between a plan that does a table scan and a plan that uses one or more nonclustered indexes. The number of logical I/O operations required for a scan of the table is equal to the number of pages in the table. A table scan starts at the first page and uses the Index Allocation Map (IAM) to access all the pages in the table. As the pages are read, the rows are evaluated to see whether they qualify based on the search criteria. Clearly, if the table from the previous example had fewer than nine total data pages, fewer logical reads would be needed to scan the whole table than to drive the scan off the nonclustered index, which was estimated to take nine logical reads.

The estimated logical reads for scanning qualifying rows is summarized in [Table 4-1](#). The access method with the least estimated cost is chosen based on this information.

Table 4-1. The Cost of Data Access for Tables with Different Index Structures

Access Method	Estimated Cost (in Logical Reads) ^[*]	Description
Table scan	The total number of data pages in the table.	
Clustered index	The number of levels in the index plus the number of data pages to scan. (Data pages to be scanned = number of qualifying rows/rows per data page.)	
Nonclustered index seek on a heap	The number of levels in the index plus the number of leaf pages plus the number of qualifying rows (a logical read for the data page of each qualifying row). The same data pages are often retrieved (from cache) many times, so the number of logical reads can be much higher than the number of pages in the table.	
Nonclustered index seek on a table with a clustered index	The number of levels in the index plus the number of leaf pages plus the number of qualifying rows times the cost of searching for a clustered index key.	
Covering nonclustered index	The number of levels in the index plus the number of leaf index pages (qualifying rows/rows per leaf page). The data page need not be accessed because all necessary information is in the index key.	

[*] All computations are rounded up to the next integer.

There is another type of index access that is combination of a nonclustered index scan and table lookup. Prior to SQL Server 2000, nonclustered index scans were only performed in the case of covering indexes, for which

all the data a query needed could be found in the keys of a nonclustered index. In this case, SQL Server never needed to access the actual data pages at all. However, in SQL Server 2000 and SQL Server 2005 a new index access algorithm was added. Suppose you have a composite index on multiple columns and one of the nonleading columns is very selective. As an example, let's use the Person.Contact table in the AdventureWorks database and build a nonclustered index on the LastName and FirstName columns as shown:

```
USE AdventureWorks
CREATE INDEX NameIndex ON Person.Contact (LastName, FirstName)
```

A query that specifies only values for the FirstName column cannot use the index to directly seek to the appropriate rows, so a nonclustered index seek won't be chosen by the optimizer. However, if the table has more than a hundred pages or so, a scan might be too expensive. Another alternative is to scan the entire nonclustered leaf level, which contains all the LastName, FirstName combinations, and when SQL Server finds the FirstName value it is looking for, it can then look up (either by RID or clustered key depending on whether the table is a heap or not) to the underlying table. This solution is only beneficial if there are very few qualifying rows. For example, the following query will return 1,255 rows and the optimizer will choose a plan using a clustered index scan:

```
SELECT * FROM Person.Contact
WHERE FirstName like 'K%'
```

However, searching for a less common value, as in the following query, yields a different plan.

```
SELECT * FROM Person.Contact
WHERE FirstName like 'Y%'
```

This query returns only 37 rows, and if you examine the plan, you'll see that SQL Server will perform a scan on the NameIndex nonclustered index, and a key lookup into the clustered index for each of the 37 qualifying rows.

Remember that SQL Server only keeps a histogram for the leading column of an index. So how would the optimizer know that there were only a very few rows where the FirstName values starts with 'Y'? This is one of the places where column statistics can be useful. Statistics on the FirstName column will let the optimizer know approximately how many rows will satisfy the condition in the WHERE clause, even though there is no index to allow a direct seek to the specific FirstName values.

Using Multiple Indexes

The query optimizer can decide to use two or more nonclustered indexes to satisfy a single query. When more than one index is considered useful because of two or more SARGs, the cost estimate changes a bit from the formulas given in [Table 4-1](#). In addition to the I/O cost of finding the qualifying rows in each index, there is an additional cost of finding the intersection of the indexes so that SQL Server can determine which data rows satisfy all search conditions. Since each nonclustered index key contains a locator for the actual data, the results of the two index searches can be treated as worktables and joined together on the locator information.

Suppose we have the following query and we have separate nonclustered indexes on both the LastName and FirstName columns. In this case, we have a clustered index on EmployeeID, which is a unique value.

Code View:

```
USE Northwind2
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName BETWEEN 'Smith' AND 'Snow' AND FirstName BETWEEN 'Daniel' and 'David'
```

If the query optimizer decides to use both indexes, SQL Server will build two worktables with the results of each index search:

Index_Key (LastName)	Bookmark (Clustered Key)
Smith	66-712403
Smith	87-120191
Smith	76-137027
Smyth	11-983770
Snow	43-220191
Snow	21-780863

Index_Key (FirstName)	Bookmark (Clustered Key)
Daniel	11-983770
Daniel	77-321456
Danielle	66-712403
David	29-331218

These worktables are joined on the row locator information, which is unique for each row. Even if you don't declare a clustered index as unique, SQL Server adds a special uniqueifier where needed so there is always a unique row locator in every nonclustered index. You can get much more detailed information about the physical storage of your indexes in Inside SQL Server 2005: The Storage Engine. From the information above, we can see that only two rows have both their first name and last name values in the requested range. The result set is:

FirstName	LastName	EmployeeID
Daniel	Smythe	11-983770
Danielle	Smith	66-712403

In this example, the data in the two indexes is enough to satisfy the query. If we also want to see information such as address and phone number, the query optimizer uses the bookmark to find the actual data row by traversing the clustered index. It can also use multiple indexes if a single table has multiple SARGs that are OR'd together. In this case, SQL Server finds the UNION of the rows returned by each index to determine all the qualifying rows. You can see the main differences if you take the same query and data and write an OR query instead of an AND:

Code View:

```
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName BETWEEN 'Smith' AND 'Snow' OR FirstName BETWEEN 'Daniel' AND 'David'
```

SQL Server can use the two indexes in the same way and build the same two worktables. However, it must UNION the two worktables together and remove duplicates. Once it removes the duplicates, it uses the

bookmarks to find the rows in the table. All the result rows will have either a first name or a last name in the requested range, but not necessarily both. For example, we can return a row for Daniel Delaney or Bridgette Smith. If we don't remove duplicates, we end up with Daniel Smythe and Danielle Smith showing up twice because they meet both conditions of the SARGs.

Unusable Statistics

In two cases, the query optimizer can't use the statistics to estimate how many rows will satisfy a given SARG. First, if there are no statistics available, SQL Server obviously can't come up with any cost estimate. However, this situation is rare in SQL Server 2005 because by default every database has the two database options, AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS, set to ON. You can turn this behavior off at the index level or the database level, but this is usually not recommended. The second situation in which there are no usable statistics is when the values in a SARG are variables. Consider this example:

```
DECLARE @name varchar(30)
SET @name = 'Zelda'
SELECT FirstName, LastName, EmployeeID FROM Employees
WHERE LastName > @name
```

Compilation and optimization takes place a batch at a time. When the SELECT query in preceding batch is optimized, the SET statement has not yet been executed and the value of @name is unknown. No specific value can be used to compare the steps in the index's statistics information, so the query optimizer must guess what the value will be.

Note



Don't confuse variables with parameters even though their syntax is nearly identical. A variable's value is never known until the statement is actually executed; it is never known at compile time. Stored procedure parameters are known when the procedure is compiled because compilation and optimization don't even take place until the procedure is actually called with specific values for the parameters.

If the query optimizer can't use the statistics for either of the reasons mentioned, the server uses fixed percentages to estimate how many rows will satisfy a given SARG. These percentages, shown here, depend on the comparison operator used. These fixed percentages can be grossly inaccurate for your specific data, however, so make sure that your statistics are up to date and usable whenever possible.

Operator	Percentage of Rows
=	10
>	30
<	30
BETWEEN	9

When the SARG involves an equality, the situation is usually handled in a special way. The 10 percent estimate shown above is actually used only in the unusual case in which there are no statistics at all. If there are statistics, they can be used. Even if we don't have a particular value to compare against the steps in the

statistics histogram, the density information can be used. That information basically tells the query optimizer the expected number of duplicates, so when the SARG involves looking for one specific value, it assumes that the value occurs the average number of times.

An even more special case occurs when the query optimizer recognizes an equality in the WHERE clause and the index is unique. Because this combination yields an exact match and always returns at most one row, the query optimizer doesn't have to use statistics. For queries of one row that use an exact match such as a lookup by a primary key, a unique nonclustered index is highly efficient. In fact, many environments probably shouldn't "waste" their clustered index on the primary key. If access via the complete primary key is common, it might be beneficial to specify NONCLUSTERED when you declare the primary key and save the clustered index for another type of access (such as a range query) that can benefit more from the clustered index.

Detecting Problems in Plans

If you've read [Chapter 3](#), you should have a basic understanding of how to read query plans and understand most of the basic operators. Now, we'll discuss some of the problems and pitfalls that you may encounter. Keep in mind that there is no one-size-fits-all approach to diagnosing problems with query plans. Queries do not perform as expected for many different reasons. Each situation is unique, and there are few if any absolute rights or wrongs. An operator may be the best choice in one scenario and a poor choice in another. Moreover, before we can evaluate whether a query plan is good or bad, we also need to understand what our goal is or what the problem is that we are trying to solve. For example, we might be trying to reduce the response time of an individual query, to improve the concurrency or throughput of the system, or to reduce the memory overhead of a query. A query plan that meets one of these goals may or may not succeed in meeting the others. In fact, in many cases, there may not be a query plan that meets all of these goals. It is often necessary to accept trade-offs, such as response time versus throughput when tuning a query or application.

Cardinality Estimation Errors

One of the first clues that a query plan may have a problem is cardinality estimation errors. It is easy to spot these errors by executing the query with SET STATISTICS PROFILE ON or SET STATISTICS XML ON. You can also examine the graphical execution plan (not the estimated plan if you want to see the actual number of rows) in SQL Server Management Studio. Simply compare the estimated and actual number of rows. There will almost always be some error in any real-world environment. The magnitude of the error is generally more important than the absolute amount of the error. For example, estimated and actual values of 1 versus 1,000, respectively, are a far bigger concern than estimated and actual values of 100,000 versus 200,000.

Cardinality errors tend to ripple or propagate upward through a plan. For example, an error that originates with a scan or seek may impact the cardinality estimates for all operators that consume the output of the scan or seek, including filters, joins, aggregations, and so on. Such an error may result in poor choices for multiple operators throughout the query plan. Assuming that you find a significant cardinality estimation error, the first step is to trace it back to the originating operator.

If the cardinality error originates from a scan or seek operator, the problem may be caused by poor statistics, an overly complex predicate, or correlations. You may be able to address the problem by creating better statistics. Consider updating statistics with a bigger sample or even a full scan. If you have a predicate that references multiple columns, create multicolumn statistics. If you have a complex predicate, try to simplify the predicate. In particular, try to use predicates in the form of SARGs, as described previously in the section "[Query Analysis](#)." Although the optimizer can sometimes derive accurate cardinality estimates for some more complex predicates, the simpler the predicate, the more likely that the estimate will be accurate. Expressing predicates in this form also increases the chances of the optimizer choosing index seeks. If you cannot simplify the predicate, consider whether you can create a computed column and create statistics on the

computed column.

If the cardinality error originates from a join, the problem may be caused by an overly complex join predicate or by a correlation between two tables. For example, suppose we are joining a customer table to an orders table with a predicate on the customers table. If, as is highly likely, some customers place more orders than others, there is no way for the optimizer to determine correctly the cardinality of the orders table. In general, if a cardinality error originates from a nonleaf operator in the query plan, chances are that you will not be able to correct it easily. In these cases, if the cardinality error is leading to a poor plan choice, you may need to use hints. Hints are discussed later in this chapter and in [Chapter 5](#). You might also be able to materialize a partial query result into a temporary table so that the optimizer can collect statistics before compiling the remainder of the query. Note that if you include both partial queries in a single batch or procedure, you may also need to force a recompile of the second query using the RECOMPILE query hint (discussed in [Chapter 5](#)).

Miscellaneous Warning Signs

Here are some additional issues to look out for.

- Watch out for nested loops joins that include bookmark lookups with high cardinalities. Nested loops joins are generally not very efficient for very large joins. Be especially on the lookout for nested loops joins with large numbers of rows on the outer side of the join. Such joins result in many executions of the inner side of the join. Even if the inner side of the join is reasonably inexpensive, with enough executions the cost can add up.
- If a query plan has memory-consuming operators such as a sort, hash join, or hash aggregate, use the SQL Profiler Sort Warning and Hash Warning event classes (in the Errors and Warnings event category) to check for spilling, which may indicate a performance problem caused by insufficient memory.
 1. In the case of sorts, especially watch out for a Sort Warning event with an EventSubClass of 2, which indicates that the sort required multiple passes and repeatedly spilled the same data to disk. This event subclass indicates an especially inefficient sort.
 2. In the case of hash operators, watch out for Hash Warning events that indicate multiple levels of hash recursion (in other words, spilling of the same data repeatedly) or hash bailout. An EventSubClass of 0 indicates normal hash recursion (in other words, normal spilling). For normal recursion, the IntegerData column indicates the recursion level or the number of times that the same data was spilled. An EventSubClass of 1 indicates that the hash operator reached the maximum recursion level and switched to the bailout algorithm.
- If you have locking, concurrency, or deadlock issues, consider whether a different index path might result in better concurrency and/or less lock contention.
- Here are some tips for OLTP workloads with strict response time requirements, high concurrency requirements, and/or high overall server throughput requirements:
 1. Watch out for blocking and/or memory-consuming operators such as sorts, hash joins, and hash aggregates. Memory-consuming operators degrade response times because of the overhead of acquiring a memory grant, and they reduce concurrency by creating contention for memory, which is a limited resource.
 2. Avoid scans and be sure that queries make maximum use of indexes. Keep in mind that the cost of a scan is proportional to the size of the table, whereas the cost of an index seek is proportional to the number of rows returned. Thus, seeks yield more consistent performance as data sizes grow.
- Here are some tips for data warehousing workloads where the response time of a single query or report is most important:
 1. Make sure that large queries that process large datasets are choosing parallel query plans.

While parallel plans do add extra query processing overhead, they also minimize the response time of queries.

2. Watch out for skew in parallel plans. Parallel scans over small tables and nested loops joins with very few rows on the outer side of the join can lead to skew problems in which some threads process many more rows or perform much more work than other threads.
3. Avoid sort-based operators, such as stream aggregates and merge joins in highly parallel plans on large machines with more processors or more cores. Specifically look out for merging exchanges at high degrees of parallelism. Merging exchanges use more server resources and are more subject to skew and scalability issues.

Monitoring Query Performance

Before you think about taking some action to make a query faster, such as adding an index or denormalizing, you should understand how a query is currently being processed. You should also get some baseline performance measurements so you can compare behavior both before and after making your changes. SQL Server provides these tools (SET options) for monitoring queries:

- STATISTICS IO
- STATISTICS TIME
- STATISTICS PROFILE
- STATISTICS XML

You enable any of these SET options before you run a query, and they will produce additional output. STATISTICS PROFILE and STATISTICS XML were described in [Chapter 3](#); STATISTICS IO and STATISTICS TIME will be explained below. Typically, you run your query with these options set in a tool such as the query window in SQL Server Management Studio. When you're satisfied with your query, you can cut and paste it into your application or into the script file that creates your stored procedures. If you use the SET commands to turn these options on, they apply only to the current connection. SQL Server Management Studio provides check boxes for turning most of these options on or off for all connections. You can go to the Tools | Options menu, then expand the Query Execution node, choose SQL Server, and then Advanced.

STATISTICS IO

Don't let the term statistics fool you. STATISTICS IO doesn't have anything to do with the statistics used for storing histograms and density information. This option provides statistics on how much work SQL Server did to process your query. When this option is set to ON, you get a separate line of output for each query in a batch that accesses any data objects. (You don't get any output for statements that don't access data, such as PRINT, SELECT the value of a variable, or call a system function.) The output from SET STATISTICS IO ON includes the values Logical Reads, Physical Reads, Read Ahead Reads, and Scan Count.

Logical Reads

This value indicates the total number of page accesses needed to process the query. Every page is read from the data cache, whether or not it was necessary to bring that page from disk into the cache for any given read. This value is always at least as large as, and is usually larger than, the value for Physical Reads. The same page can be read many times (such as when a query is driven from an index), so the count of Logical Reads for a table can be greater than the number of pages in a table.

Physical Reads

This value indicates the number of pages that were read from disk; it is always less than or equal to the value of Logical Reads. The value of the Buffer Cache Hit Ratio, as displayed by System Monitor, is computed from the Logical Reads and Physical Reads values as follows:

$$\text{Cache-Hit Ratio} = (\text{Logical Reads} - \text{Physical Reads})/\text{Logical Reads}$$

Remember that the value for Physical Reads can vary greatly and decreases substantially with the second and subsequent execution because the cache is loaded by the first execution. The value is also affected by other SQL Server activity and can appear low if the page was preloaded by read ahead activity. For this reason, you probably won't find it useful to do a lot of analysis of physical I/O on a per-query basis. When you're looking at individual queries, the Logical Reads value is usually more interesting because the information is consistent. Physical I/O and achieving a good cache-hit ratio is crucial, but they are more interesting at the server level. Pay close attention to Logical Reads for each important query, and pay close attention to physical I/O and the cache-hit ratio for the server as a whole.

STATISTICS IO acts on a per-table, per-query basis. You might want to also examine some of the columns keeping track of physical reads in the sys.dm_exec_query_stats DMV. This information is cumulative for each query plan, and contains columns such as min_physical_reads, max_physical_reads, and total_physical_reads for all executions of each query. You could also look at the reads value in sys.dm_exec_sessions to get per-session information.

Read Ahead Reads

The Read Ahead Reads value indicates the number of pages that were read into cache using the read ahead mechanism while the query was processed. These pages are not necessarily used by the query. If a page is ultimately needed, a logical read is counted but a physical read is not. A high value means that the value for Physical Reads is probably lower and the cache-hit ratio is probably higher than if a read ahead was not done. In a situation like this, you shouldn't infer from a high cache-hit ratio that your system can't benefit from additional memory. The high ratio might come from the read ahead mechanism bringing much of the needed data into cache. That's a good thing, but it might be better if the data simply remains in cache from previous use. You might achieve the same or a higher cache-hit ratio without requiring the Read Ahead Reads.

You can think of read ahead as simply an optimistic form of physical I/O. In full or partial table scans, the table's IAMs are consulted to determine which extents belong to the object. The extents are read with a single 64-KB scatter read, and because of the way that the IAMs are organized, they are read in disk order. If the table is spread across multiple files in a file group, the read ahead attempts to keep at least eight of the files busy instead of sequentially processing the files. Read Ahead Reads are asynchronously requested by the thread that is running the query; because they are asynchronous, the scan doesn't block while waiting for them to complete. It blocks only when it actually tries to scan a page that it thinks has been brought into cache and the read hasn't finished yet. In this way, the read ahead neither gets too ambitious (reading too far ahead of the scan) nor too far behind.

Scan Count

The Scan Count value indicates the number of times that the corresponding table was accessed. Outer tables of a nested loop join have a Scan Count of 1. For inner tables, the Scan Count might be the number of times "through the loop" that the table was accessed. The number of Logical Reads is determined by the sum of the Scan Count times the number of pages accessed on each scan. However, even for nested loop joins, the Scan Count for the inner table might show up as 1. SQL Server might copy the needed rows from the inner table into a worktable in cache and use this worktable to access the actual data rows. When this step is used in the

plan, there is often no indication of it in the STATISTICS IO output. You must use the output from STATISTIC TIME, as well as information about the actual processing plan used, to determine the actual work involved in executing a query. Hash joins and merge joins usually show the Scan Count as 1 for both tables involved in the join, but these types of joins can involve substantially more memory. You can inspect the granted_query_memory value in sys.dm_exec_requests while the query is being executed, but keep in mind that this is not a cumulative counter and is valid only for the currently running query. Alternatively, you can examine the memory_usage column in sys.dm_exec_sessions to see the number of 8KB pages of memory used by each session.

Troubleshooting Example Using STATISTICS IO

The following situation is based on a real-world problem encountered when doing a simple SELECT * from a table with only a few thousand rows. The amount of time this simple SELECT took seemed to be out of proportion to the size of the table. We can create a similar situation here by making a copy of the orders table in the Northwind2 database and adding a new column to it with a default value of 2,000-byte character field.

```
SELECT *
INTO neworders
FROM Orders
GO
ALTER TABLE neworders
ADD full_details char(2000) NOT NULL DEFAULT 'full details'
```

We can gather some information about this table with the following command:

```
EXEC sp_spaceused neworders, true
```

The results will tell us that there are 830 rows in the table, 3,272 KB for data, which equates to 409 data pages. That doesn't seem completely unreasonable, so we can try selecting all the rows after enabling STATISTICS IO:

```
SET STATISTICS IO ON
GO
SELECT * FROM neworders
```

The results now show that SQL Server had to perform 1,179 logical reads for only 409 data pages. There is no possibility that a nonclustered index is being used inappropriately because we just created this table, and there are no indexes at all on it. So where are all the reads coming from? In Inside SQL Server 2005: The Storage Engine, the author discussed the fact that if a row is increased in size so that it no longer fits on the original page, it will be moved to a new page and SQL Server will create a forwarding pointer to the new location from the old location. Note that this is not the same as row-overflow data, which occurs when individual rows are too long to fit on a page. In this situation, the rows are all less than the maximum of 8,060 bytes, but the pages the rows are on do not have enough free space to contain some of the rows after the additional 2,000-byte columns are added. We can use the Dynamic Management Function sys.dm_db_index_physical_stats to observe how many forwarded rows exist in the neworders table, using the following query:

```
SELECT forwarded_record_count
FROM sys.dm_db_index_physical_stats
(db_id('Northwind2'), object_id('neworders'), null, null, 'detailed')
```

This query returns a value of 770. If you play around with the numbers a bit, you might realize that $770 + 409$ equals 1,179, the number of reads we ended up performing with the table scan. We then realized that when scanning a table, SQL Server will read every page (there are 409 of them) and each time a forwarding pointer is encountered, SQL Server will jump ahead to that page to read the row. So for each forwarding pointer, there is one additional logical reads. SQL Server is doing a lot of work here for a simple table scan.

The "fix" to this problem is to regularly check the number of forwarding pointers, especially if a table has variable-length fields and is frequently updated. If you notice a high number of forwarding pointers, you can rebuild the table by creating a clustered index. If for some reason we don't want to keep the clustered index, we can drop it. But the act of creating the clustered index will reorganize the table.

Code View:

```
SET STATISTICS IO OFF
GO
CREATE CLUSTERED INDEX orderID_cl ON neworders(orderID)
GO
DROP INDEX neworders.orderID_cl
GO
EXEC sp_spaceused neworders, true
SET STATISTICS IO ON
GO
SELECT ROWS = count(*) FROM neworders

RESULTS:
name           rows      reserved      data      index_size      unused
-----        -----      -----      -----      -----      -----
neworders       830        2248 KB     2216 KB      8 KB          24 KB

ROWS
-----
830
(1 row(s) affected)
Table 'neworders'. Scan count 1, logical reads 277, physical reads 0, read-ahead reads 0.
```

This output shows us that not only did the building of the clustered index reduce the number of pages in the table, from 409 to 277, but the cost to read the table is now the same as the number of pages in the table.

STATISTICS TIME

The output of SET STATISTICS TIME ON is pretty self-explanatory. It shows the elapsed time and CPU time required to process the query. In this context, CPU time means the time not spent waiting for resources such as locks or reads to complete. The times are separated into two components: the time required to parse and compile the query, and the time required to execute the query. For some of your testing, you might find it just as useful to look at the system time with getdate before and after a query if all you need to measure is elapsed time. However, if you want to compare elapsed time vs. actual CPU time or if you're interested in how long compilation and optimization took, you must use STATISTICS TIME.

In many cases, you'll notice that the output includes two sets of data for parse and compile time. This happens when the plan is being added to the plan cache for possible reuse. The first line is the actual parse and compile before placing the plan in cache, and the second line appears when SQL Server is retrieving the plan from cache. Subsequent executions will still show the same two lines, but if the plan is actually being reused, the parse and compile time for both lines will be 0.

Query Improvements

Many techniques are available for improving the performance of your queries. Some of the techniques are no more complicated than simply rewriting some of your Transact-SQL code, while others involve some redefinitions of your database schema, verifying that appropriate statistics are available, or the creation of new indexes. In other cases, you may find that you need to supply specific information to the query optimizer in the form of hints to get the maximum possible performance from your queries. The following sections describe these possible areas of improvement.

Rewriting Your Query

Frequently, you can write the same query in many ways. As we've seen, these rewrites often result in different—sometimes better and sometimes worse—query plans. Here are a few ideas for improving queries:

- Besides avoiding overly complex predicates, which as we've discussed may lead to cardinality errors and bad plans, also try to limit your use of outer joins, cross apply, outer apply, correlated scalar subqueries, or other excessively complex subqueries. These constructs have a tendency to limit the optimizer's choice of join order and join type. The optimizer may not be able to reorder joins in a query plan with outer joins. The optimizer may be forced to use nested loops joins, regardless of the performance implications for queries with excessively complex cross applies or subqueries. At the very least, avoid scalar subqueries if there is a semantically equivalent nonscalar subquery and if you do not need SQL Server to validate that the subquery produces exactly one row.
- In some cases, it may help to evaluate a noncorrelated scalar subquery as a separate stand-alone query, save the results in a Transact-SQL variable, and then use that variable in a subsequent query. With a single query, the optimizer cannot know the results of the subquery and, depending on the query, the data, and the available statistics, may generate poor cardinality estimates for the remainder of the plan. However, by computing the results of the subquery separately, the optimizer can use the result when compiling the remainder of the query and may generate more accurate cardinality estimates and a better plan. If you include both queries in a single batch or procedure, you may need to force a recompile of the second query.
- Rewriting noncorrelated scalar subqueries as multiple queries can also help with parallel plans. The optimizer frequently implements such subqueries using a nested loops join with the single row result of the subquery as the outer input to the join. In a parallel plan, a nested loops join with a single outer row effectively executes as a serial plan. By rewriting the query in two stages, the optimizer may be able to generate a plan that can take better advantage of parallelism.
- Try to avoid dynamic index seeks in parallel plans. The optimizer tends to choose dynamic index seeks (discussed in [Chapter 3](#)) for parameterized queries with OR'ed predicates or IN lists. The nested loops join used by dynamic index seeks tends to limit parallelism because of the low number of rows on the outer side of the join. If the table in question has a unique key, one way to avoid dynamic index seeks is to rewrite the query as a UNION. In some cases, such a rewrite may result in a better plan. Another way to avoid dynamic index seeks is to rewrite queries without parameters. By using literals in a query, the optimizer can eliminate duplicates at compile time and generate a static index seek. On the other hand, it is generally a good idea to avoid dynamic SQL and to make use of parameterized queries or stored procedures. Parameterized queries and stored procedures are generally more secure against SQL injection attacks, make more effective use of the plan cache, and reduce compilation time overhead. So, be very careful when substituting literals for parameters.
- Use inline table-valued functions (TVFs) instead of multistatement TVFs whenever possible. Especially try to avoid using cross apply with parameterized multistatement TVFs. SQL Server expands inline TVFs into the main query much like it expands views but evaluates multistatement TVFs in a separate context from the main query. Moreover, the server materializes the results of multistatement TVFs into temporary worktables. The separate context and worktable make multistatement TVFs costly. If a multistatement TVF is combined with a cross apply, SQL Server

- invokes the TVF and populates the worktable multiple times (once for each row generated by the outer input to the cross apply). Thus, the cross apply further inflates the cost of executing the TVF.
- Try to stick to set-based queries and avoid cursors. Set-based queries are nearly always more efficient than equivalent cursor-based alternatives. Moreover, it is possible to express many surprisingly complex queries without the use of cursors. Do not assume that a cursor is the only option. If you do use cursors, be careful with dynamic cursors. Dynamic cursors significantly limit the choice of plans available to the query optimizer. For example, dynamic cursors limit the optimizer to using nested loops joins. In some cases, choosing a dynamic cursor induces the optimizer to choose a far worse overall plan. If in doubt, compare the query plan for a dynamic cursor with the query plan for the same query submitted without a dynamic cursor. If there is a significant difference between the dynamic and nondynamic plans, make sure that the differences are not harming performance. Also consider whether you really need a dynamic cursor or whether you could use a less restrictive cursor type such as a static or keyset cursor.

Schema Improvements

In some cases, minor schema changes can lead to dramatically better plans. Here are some ideas worth considering:

- Do you have a missing index or an index with a missing column? If you have an index or table scan with a highly selective predicate, consider building an appropriate index so that the optimizer can choose an index seek instead of a scan. If you have a plan that would be a good candidate for an index nested loops join, be sure that you have the appropriate index for the table on the inner side of the join. If you have a plan with an expensive bookmark lookup (that is, one that fetches many rows), consider whether you can create an index that covers the columns in question. Remember that in SQL Server 2005, you can include nonkey columns in a nonclustered index. The Database Engine Tuning Advisor (DTA), formerly the Index Tuning Wizard (ITW), may help identify missing indexes. SQL Server 2005 also includes the new missing indexes DMVs (including `sys.dm_db_missing_index_details` and `sys.dm_db_missing_index_group_stats`), which may help. However, bear in mind that SQL Server must maintain all indexes, so indexes on frequently updated columns (or indexes that include such columns) could lead to degraded insert, update, or delete performance. More information on building appropriate indexes is available in the section on "[Creating Useful Indexes](#)."
- Would creating an index eliminate a sort operator? If you have (or want) a plan with a merge join or a stream aggregate and want to eliminate a sort operator, consider whether you might be able to create an index to provide the necessary sort order. If you create a new index, don't forget that the index must cover the appropriate columns or the optimizer may not choose it. Also, whenever possible, create unique indexes or constraints. By creating the appropriate unique indexes, the optimizer can make smarter choices such as using a one-to-many merge join instead of a many-to-many merge join. Be sure to use the minimum set of key columns for each unique index and add additional columns, if any, as included columns. Minimizing the set of keys creates the "narrowest" possible uniqueness constraint.
- Consider creating appropriate FOREIGN KEY, NOT NULL, and CHECK constraints. Foreign key constraints can help simplify joins by converting some outer or semi-joins to inner joins (or a full outer join to a left or right outer join). In some cases, a foreign key constraint in conjunction with a NOT NULL constraint can eliminate a join entirely if the data from the primary key table is only required to validate that a row exists. Check constraints can also help a bit by eliminating unnecessary or redundant predicates.
- Try to avoid joining tables on columns with mismatched data types. In particular, try to avoid creating primary and foreign keys with mismatched data types. For example, do not create a numeric primary key and an integer foreign key. More importantly, do not create a numeric (or integer or float) primary key and a string foreign key (or vice versa). Although SQL Server can perform comparisons of mismatched data types, such comparisons may introduce implicit conversions, tend to be less

efficient, may introduce undesirable semantics (especially when comparing numeric and string data), and may in some cases prevent SQL Server from using index seeks.

Statistics Management

In addition to the statistics that SQL Server maintains for indexes, SQL Server also has built-in policies to create statistics on unindexed table columns; these are frequently referred to as column statistics. By default, statistics on both indexed and unindexed columns are maintained automatically. These policies work well for most scenarios. However, in some situations it might be necessary to manage your statistics manually. We described the command to view statistics earlier in this chapter, and you should note that the same command we used for viewing index statistics can be used to view column statistics. In this section, we'll first tell you about column statistics. Then we'll describe the commands you can use to manage your statistics. Finally, we describe common situations where manual control might be appropriate. [Chapter 5](#) goes into more detail about the relationship between updating statistics and recompiling your query plans.

Column Statistics

By default, SQL Server automatically creates statistics every time an unindexed column is referenced in the WHERE clause of a query. You can see this behavior by first examining the Customers table in the Northwind2 database with the following command:

```
EXEC sp_helpstats Customers;
```

The first time you run this after creating the Northwind2 database, you should not get any statistics information returned. However, if you query the Customers table on an unindexed column and then examine the statistics again, you should see different results.

```
SELECT * FROM Customers
WHERE ContactName = 'Hanna Moos';
GO
EXEC sp_helpstats Customers;
RESULTS:
statistics_name          statistics_keys
-----                  -----
_WA_Sys_00000003_0519C6AF    ContactName
```

The results here show you the automatically created statistics, but it would also return any statistics you created using the CREATE STATISTICS command. The automatically created statistics have a system-generated name that always starts with the eight characters '_WA_Sys_'. This is followed by a hex value indicating the ordinal position of the column in the table, and a hex value representing the object_id of the table. Manually created statistics can have any name you choose as long as it is a legal identifier name. SQL Server 2005 also provides a catalog view called sys.stats to allow you to list your column statistics, but in order to get the basic information provided by the sp_helpstats procedure, you would need to join sys.stats with sys.stats_columns and sys.columns, and include special processing for the cases where manually created statistics included multiple columns. The automatically created statistics are always on a single column. Whether you want statistics to be automatically created or not can be controlled with the database option auto create statistics.

Since SQL Server also creates statistics for every index you build, to see a complete list of the statistics on your table you would also need to enumerate your indexes. You could do that using either of the following

statements:

```
EXEC sp_helpindex Customers;
```

Or

```
SELECT * FROM sys.indexes
WHERE object_id = OBJECT_ID('Customers');
```

In most cases, allowing SQL Server to manage statistics automatically is the best policy. However, there are some situations in which you might consider either adding a computed column on which SQL Server can create statistics or manually creating statistics. The following two sections cover some of these situations.

Statistics on Computed Columns

Typically, column statistics are not useful to estimate the filter selectivity on the result of scalar expressions. For example, in the Northwind2 database, the table Order Details has a column for UnitPrice and another for Quantity. A query on the total sales amount might have a filter of the following form:

```
WHERE UnitPrice * Quantity > 1000
```

If SQL Server tries to estimate the number of rows satisfying this condition using the column statistics on UnitPrice and Quantity, it is likely to encounter large estimation errors. The SQL Server 2005 optimizer uses a standard guess of 30 percent selectivity when estimating how many rows might satisfy an inequality comparison as in the previous example. Large estimation errors can produce a suboptimal query plan. Specifically, the errors will affect the choice of operations that appear in the plan after the filter itself. As an alternative, you can define a computed column for the expression on which you want to filter, using a command like the following:

```
ALTER TABLE [Order Details]
ADD TotalSale AS UnitPrice * Quantity;
```

In SQL Server 2005, the query optimizer will detect the use of the computed column in queries, just as it does for any unindexed column reference, and it will create statistics on the computed column. The statistics will allow the optimizer to determine the appropriate cardinality estimation on the filter.

Multicolumn Statistics

Multicolumn statistics are helpful to the query optimizer when columns are correlated and processed together in queries. For example, if you have a table of locations and have columns for State and City, rows will not include all possible combination of values for the two columns. Single-column statistics have no way to represent the information that not all State and City combinations are possible. For example, City = 'Seattle' can only go with State = 'Washington.' As an alternative, you can manually create multicolumn statistics. In SQL Server 2005, multicolumn statistics consist of the number of distinct values for sets of columns. Creating statistics on columns (City, State) computes the number of distinct values for columns (City) and (City, State). In this situation, you might have 80 distinct values for (City), and 80 distinct values for (City, State), even

though you have only 30 values for State. This information is used to estimate the result of GROUP BY operations, and for filtering. For filtering, SQL Server's optimizer will ignore a condition on State if you are already filtering on City.

You create multicolumn statistics by using the following statement:

```
CREATE STATISTICS LocationStatistics
    ON Locations(City, State);
```

Multicolumn statistics are automatically created on multicolumn indexes, and cover all prefixes of the index. As mentioned earlier in this chapter, for an index on (A, B, C), SQL Server maintains density information on (A), (A, B), and (A, B, C). Unlike single-column statistics, multicolumn statistics are not automatically created for unindexed columns. Therefore, you might have to create these yourself in the case of correlated columns (like City, State) that are frequently used together for grouping or filtering purposes in your queries. Just like statistics on indexes, SQL Server will automatically update single-column and multicolumn statistics, if the database setting AUTO_UPDATE_STATISTICS is ON.

Manual Statistics Refresh

You can use the UPDATE STATISTICS command to manually update statistics and change the default sampling size. By default, SQL Server only analyzes a sample of the rows in the table when generating the statistics information. You can provide a larger sampling size in the command, including a full scan of the table. When troubleshooting bad query plans, it is a good idea to update all statistics on the tables with a full scan. The better statistics that result often correct the problem.

To refresh all statistics on the Orders table with a full scan, use the following command:

```
UPDATE STATISTICS Orders WITH FULLSCAN;
```

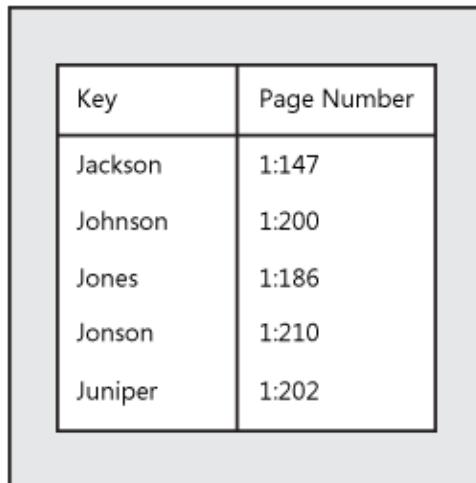
The frequency with which statistics are automatically updated depends on how much data is changing, and how often. If the impact of statistics update operations is interfering with your query performance, you may choose to disable automatic updates and perform manual updates as a regularly scheduled job. Alternatively, SQL Server 2005 provides a new database option called AUTO_UPDATE_STATISTICS_ASYNC, which won't interfere with the currently executing statement. More details about the automatic statistics update behavior, including the new AUTO_UPDATE_STATISTICS_ASYNC, is covered in [Chapter 5](#).

Creating Useful Indexes

Creating useful indexes is one of the most important tasks you can do to achieve good performance. Indexes can dramatically speed up data retrieval and selection, but they can have a negative impact on data modification operations because along with changes to the data, the index entries must also be maintained and those changes must be logged. The key to creating useful indexes is understanding the uses of the data, the types and frequencies of queries performed, and how queries can use indexes to help SQL Server find your data quickly. You might want to make sure you thoroughly understand the difference between clustered and nonclustered indexes because the difference is crucial in deciding what kind of index to create and the following discussion contains only a very brief overview. You can get useful information about the structure and basic use of indexes in two earlier volumes in this series, Inside SQL Server 2005: TSQL Querying and Inside SQL Server 2005: The Storage Engine.

Clustered and nonclustered indexes are similar at the upper (node) levels—both are organized as B-trees. Index rows above the leaf level contain index key values and pointers to pages at the next level down. Each row keeps track of the first key value on the page it points to. [Figure 4-3](#) shows an abstract view of an index node for an index on a customer's last name. The entry Johnson indicates page 1:200 (file 1, page 200), which is at the next level of the index. Since Johnson and Jones are consecutive entries, all the entries on page 1:200 have values between Johnson (inclusive) and Jones (exclusive).

Figure 4-3. An index node page



Key	Page Number
Jackson	1:147
Johnson	1:200
Jones	1:186
Jonson	1:210
Juniper	1:202

The leaf, or bottom, level of the index is where clustered and nonclustered indexes differ. For both kinds of indexes, the leaf level contains every key value in the table on which the index is built, and those keys are in logically sorted order. In a clustered index, the leaf level is the data level, so of course every key value is present. This means that the data in a table is logically sorted in the order of the clustered index keys. In a nonclustered index, the leaf level is separate from the data. In addition to the key values, the index rows contain a bookmark indicating where to find the actual data. If the table has a clustered index, the bookmark is the clustered index key that corresponds to the nonclustered key in the row. (If the clustered key is composite, all parts of the key are included.)

Clustered indexes are guaranteed to be unique in SQL Server 2005; if you don't declare them as unique, SQL Server adds a uniqueifier to every duplicate key to turn the index into a unique composite index. If our index on last name is a nonclustered index and the clustered index on the table is the zip code, a leaf-level index page might look something like [Figure 4-4](#). The number in parentheses after the zip code is the uniqueifier and appears only when there are duplicate zip codes.

Figure 4-4. A leaf-level index page

Key	Locator
Johnson	98004(1)
Johnson	06801(3)
Johnston	70118
Johnstonc	33801(2)
Johnstone	95014(2)
Johnstone	60013
Jonas	80863(2)
Jonas	37027
Jonasson	22033(4)

Choose the Clustered Index Carefully

Clustered indexes are extremely useful for range queries (for example, WHERE sales_quantity BETWEEN 500 and 1000) and for queries in which the data must be ordered to match the clustering key. Only one clustered index can exist per table, since it defines the ordering of the data for that table. Since you can have only one clustered index per table, you should choose it carefully based on the most critical retrieval operations. Because of the clustered index's role in managing space within the table, nearly every table should have one.

If a table is declared with a primary key (which is advisable), by default a clustered index will be built on the primary key columns. However, this is not always the best choice. The primary key is, by definition, unique, and nonclustered indexes are very efficient at finding a single qualifying row and can still enforce the primary key's uniqueness. So save your clustered index for something that will benefit more from it by adding the keyword NONCLUSTERED when you declare the PRIMARY KEY constraint.

Make Nonclustered Indexes Highly Selective

A query using an index on a large table is often dramatically faster than a query doing a table scan. But this is not always true, and table scans are not all inherently evil. Nonclustered index retrieval means reading B-tree entries to determine the data page that is pointed to and then retrieving the page, going back to the B-tree, retrieving another data page, and so on until many data pages are read over and over. (Subsequent retrievals can be from cache.) With a table scan, the pages are read only once. If the index does not disqualify a large percentage of the rows, it is cheaper to simply scan the data pages, reading every page exactly once.

The query optimizer greatly favors clustered indexes over nonclustered indexes, because in scanning a clustered index the system is already scanning the data pages. Once it is at the leaf of the index, the system has gotten the data as well. So there is no need to read the B-tree, read the data page, and so on. This is why nonclustered indexes must be able to eliminate a large percentage of rows to be useful (that is, highly selective), whereas clustered indexes are useful even with less selectivity.

Indexing on columns used in the WHERE clause of frequent or critical queries is often a big win, but this usually depends on how selective the index is likely to be. For example, if a query has the clause WHERE last_name = 'Stankowski', an index on last_name is likely to be very useful; it can probably eliminate 99.9 percent of the rows from consideration. On the other hand, a nonclustered index will probably not be useful on a clause of WHERE sex = 'M' because it eliminates only about half of the rows from consideration; the repeated steps needed to read the B-tree entries just to read the data require far more I/O operations than simply making one single scan through all the data. So nonclustered indexes are typically not useful on columns that do not have a wide dispersion of values.

Think of selectivity as the percentage of qualifying rows in the table (qualifying rows/total rows). If the ratio of qualifying rows to total rows is low, the index is highly selective and is most useful. If the index is used, it can eliminate most of the rows in the table from consideration and greatly reduce the work that must be performed. If the ratio of qualifying rows to total rows is high, the index has poor selectivity and will not be useful. A nonclustered index is most useful when the ratio is around 5 percent or lessâ—that is, if the index can eliminate 95 percent of the rows from consideration. If the index would return more than 5 percent of the rows in a table, it probably will not be used; either a different index will be chosen or the table will be scanned. Recall that each index has a histogram of sampled data values for the index key, which the query optimizer uses to estimate whether the index is selective enough to be useful to the query.

Tailor Indexes to Critical Transactions

Indexes speed data retrieval at the cost of additional work for data modification. To determine a reasonable number of indexes, you must consider the frequency of updates versus retrievals and the relative importance of the competing types of work. If your system is almost purely a reporting or decision-support system with little update activity, it makes sense to have as many indexes as will be useful to the queries being issued. A reporting system might reasonably have a dozen or more indexes on a single table. If you have a predominantly online transaction processing (OLTP) application, you need relatively few indexes on a tableâ—probably just a couple of carefully chosen ones.

Look for opportunities to achieve index coverage in queries, because a covering index can be even more useful than a clustered index. An index "covers" the query if it has all the data values needed as part of the index key. For example, if you have a query such as SELECT emp_name, emp_sex FROM employee WHERE emp_name LIKE 'Sm%' and you have a nonclustered index on emp_name, it might make sense to append the emp_sex column to the index key as well. Then the index will still be useful for the selection, but it will already have the value for emp_sex. The query optimizer won't need to read the data page for the row to get the emp_sex value; the query optimizer is smart enough to simply get the value from the B-tree key. The emp_sex column is probably a char(1), so the column doesn't add greatly to the key length, and this is good.

Every nonclustered index is a covering index if all you are interested in is the key column of the index. For example, if you have a nonclustered index on first name, it covers all these queries:

- Select all the first names that begin with K.
- Find the first name that occurs most often.
- Determine whether the table contains the name Melissa.

In addition, if the table also has a clustered index, every nonclustered index includes the clustering key. So it can also cover any queries that need the clustered key value in addition to the nonclustered key. For example, if our nonclustered index is on the first name and the table has a clustered index on the last name, the following queries can all be satisfied by accessing only leaf pages of the nonclustered index B-tree:

- Select Tibor's last name.
- Determine whether any duplicate first- and last-name combinations exist.
- Find the most common first name for people with the last name Wong.

You can go too far and add too many columns to the index. The net effect is that the index becomes a virtual copy of the table, just organized differently. Far fewer index entries fit on a page, I/O increases, cache efficiency is reduced, and much more disk space is required. The covered query technique can improve performance in some cases, but you should use it with discretion. In those cases where there is not one best choice for clustered index, you can think of your covering indexes like mini-clustered indexes, on a subset of the data.

A unique index (whether nonclustered or clustered) offers the greatest selectivity— that is, only one row can match— so it is most useful for queries that are intended to return exactly one row. Nonclustered indexes are great for single-row accesses via the PRIMARY KEY or UNIQUE constraint values in the WHERE clause.

Indexes are also important for data modifications, not just for queries. They can speed data retrieval for selecting rows, and they can speed data retrieval needed to find the rows that must be modified. In fact, if no useful index for such operations exists, the only alternative is for SQL Server to scan the table to look for qualifying rows. UPDATE or DELETE operations on only one row are common; you should make sure there is a unique index on the search columns used to find the rows being modified. A need to update indexed columns can affect the update strategy chosen. For example, to update a column that is part of the key of the clustered index on a table, you must process the update as a delete followed by an insert rather than as an update-in-place. When you decide which columns to index, especially which columns to make part of the clustered index, consider the effects the index will have on the update method used.

SQL Server provides a function that can give us a workaround if you need to index a large character field. The CHECKSUM function computes a checksum on a row or a column, and its value is always a 4-byte integer. You can create a computed column to be the checksum of your large character field and then build an index on that computed column. The values returned by CHECKSUM are not guaranteed to be absolutely unique, but there will be few duplicates. Since there is the possibility of two character string values having the same value for the checksum, your queries will need to include the full string that you're looking for.

Here's an example. Suppose we're expecting the titles table in the pubs database to grow to over a million titles by the end of the year. We want to be able to do quick lookups on book names, but since the title field is 80 characters wide, we know that an index on title is not ideal. (In fact, the titles table does have an index on title, but since it's not really such a good idea, we'll drop it.) So we'll create a computed column using CHECKSUM and index that:

```
USE pubs
GO
DROP INDEX titles.titleind
GO
ALTER TABLE titles
ADD hash_title AS CHECKSUM(title)
GO
CREATE INDEX hash_index ON titles(hash_title)
GO
```

First, let's try just searching for a particular title:

```
SELECT * FROM titles
WHERE title = 'Cooking with Computers: Surreptitious Balance Sheets'
```

If you look at the query plan for this SELECT, you'll see that a clustered index scan is done, which means the whole table must be searched. Instead, let's also query on the checksum value in the hash_title column:

Code View:

```
SELECT * FROM titles
WHERE title = 'Cooking with Computers: Surreptitious Balance Sheets'
AND hash_title = CHECKSUM('Cooking with Computers: Surreptitious Balance Sheets')
```

The query plan will now show that SQL Server can do an index seek on the computed column hash_title. We'll look at query plans in detail later in this chapter.

Pay Attention to Column Order

At the risk of stating the obvious, an index can be useful to a query only if the criteria of the query match the columns that are leftmost in the index key. For example, if an index has a composite key of (last_name, first_name), that index is useful for a query such as WHERE last_name = 'Smith' or WHERE last_name = 'Smith' AND first_name = 'John'. But it may not be useful for a query such as WHERE first_name = 'John'. Think of using the index like a phone book. You use a phone book as an index on last name to find the corresponding phone number. But the standard phone book is useless if you know only a person's first name because the first name might be located on any page.

In general, you should put the most selective columns leftmost in the key of nonclustered indexes, or at least, not put extremely nonselective columns first. For example, an index on emp_name, emp_sex is useful for a clause such as WHERE emp_name = 'Smith' AND emp_sex = 'M'. But if the index is defined as emp_sex, emp_name, it isn't useful for most retrievals. The leftmost key, emp_sex, cannot rule out enough rows to make the index useful. Be especially aware of this when it comes to indexes that are built to enforce a PRIMARY KEY or UNIQUE constraint defined on multiple columns. The index is built in the order that the columns are defined for the constraint. So you should adjust the order of the columns in the constraint to make the index most useful to queries; doing so will not affect its role in enforcing uniqueness.

There are always exceptions to every rule of putting your most selective column leftmost. Earlier in the chapter, we mentioned a situation in which it could be useful to not put a very selective column in the leftmost position. We mentioned a situation in which, if SQL Server knew there were only one or two rows with a value that was not the first value of the index, SQL Server could perform a leaf-level scan and an index lookup on a small number of rows.

Index Columns Used in Joins

Index columns are frequently used to join tables. When you create a PRIMARY KEY or UNIQUE constraint, an index is automatically created for you. But no index is automatically created for the referencing columns in a FOREIGN KEY constraint. Such columns are frequently used to join tables, so they are almost always among the most likely ones on which to create an index. If your primary key and foreign key columns are not naturally compact, consider creating a surrogate key using an identity column (or a similar technique). As with row length for tables, if you can keep your index keys compact, you can fit many more keys on a given page, which results in less physical I/O and better cache efficiency. And if you can join tables based on integer values such as an identity, you avoid having to do relatively expensive character-by-character comparisons. Ideally, columns used to join tables are integer columns.

We talked about density in the statistics discussion earlier in the chapter. When determining the density of a search argument (SARG), the query optimizer can look up a specific value in the statistics histogram to get an estimate of the number of occurrences. However, when it looks at whether an index is useful for a join operation, there is no specific value that it can look up. The query optimizer needs to know how many rows in one table are likely to match the join columns value from the other table. In other words, if two tables are

related in a one-to-many relationship, how many is "many"? We use the term join density to mean the average number of rows in one table that match a row in the table it is being joined to. You can also think of join density as the average number of duplicates for an index key. A column with a unique index has the lowest possible join density (there can be no duplicates) and is therefore extremely selective for the join. If a column being joined has a large number of duplicates, it has a high density and is not very selective for joins.

As you learned in [Chapter 3](#), joins are frequently processed as nested loops. For example, if while joining the orders table with order details the system starts with the orders table (the outer table), a nested loops join would take each qualifying row in orders and search the inner table (order details) for matching rows. Think of the join being processed as, "Given a specific row in the outer table, go find all corresponding rows in the inner table." If you think of joins in this way, you'll realize that it's important to have a useful index on the inner table, which is the one being searched for a specific value. For the most common type of join, an equijoin that looks for equal values in columns of two tables, the query optimizer automatically decides which is the inner table and which is the outer table of a join. The table order that you specify for the join doesn't matter in the equijoins case. However, the order for outer joins must match the semantics of the query, so the resulting order is dependent on the order specified.

Create or Drop Indexes as Needed

If you create indexes but find that they aren't used, you should drop them. Unused indexes slow data modification without helping retrieval. You can determine whether indexes are used by watching the plans produced via the SHOWPLAN options. However, this isn't easy if you're analyzing a large system with many tables and indexes. There might be thousands of queries that can be run and no way to run and analyze the SHOWPLAN output for all of them. An alternative is to use the Database Engine Tuning Advisor (DTA) to generate a report of current usage patterns. This tool is designed to determine which new indexes to build, but you can use it simply as a reporting tool to find out what is happening in your current system. We'll look at the DTA in the next section.

Some batch-oriented processes that are query intensive can benefit from certain indexes. Such processes as complex reports or end-of-quarter financial closings often run infrequently. If this is the case, remember that creating and dropping indexes is simple. Consider a strategy of creating certain indexes in advance of your batch processes and then dropping them when those batch processes are done. In this way, the batch processes benefit from the indexes but do not add overhead to your OLTP usage.

The Database Engine Tuning Advisor

You've probably seen cases where the query optimizer can come up with a reasonably effective query plan even in the absence of well-planned indexes on your tables. However, this does not mean that a well-tuned database doesn't need good indexes. The query plans that don't rely on indexes frequently consume additional system memory, and other applications might suffer. In addition, having appropriate indexes in place can help solve many blocking problems.

Designing the best possible indexes for the tables in your database is a complex task; it not only requires a thorough knowledge of how SQL Server uses indexes and how the query optimizer makes its decisions, but it requires that you be intimately familiar with how your data will actually be used. The SQL Server 2005 Database Engine Tuning Advisor (DTA), mentioned briefly earlier in the chapter, is a powerful tool for helping you design the best possible indexes.

You can access the DTA from the Tools menu in SQL Server Management Studio or from the Tools menu in SQL Server Profiler. You can also access it from the Start Menu, by clicking on Microsoft SQL Server 2005, then Performance Tools, then Database Engine Tuning Advisor. The DTA can tune multiple databases at a time, and it bases its recommendations on a workload file that you provide. The workload file can be a file of captured trace events that contains at least the RPC and SQL Batch starting or completed events, as well as the

text of the RPCs and batches. It can also be a file of SQL statements. If you use SQL Profiler to create the workload file, you can capture all SQL statements submitted by all users over a period of time. The DTA can then look at the data access patterns for all users, for all tables, and can make recommendations with everyone in mind. In addition to recommending clustered and nonclustered indexes to create, the DTA can also recommend indexed views and table partitioning, if it determines that those structures can improve the performance of the queries in your workload.

Optimization Hints in SQL Server 2005

In this section, we'll look at hints that can be used to impact the query plan selected by the optimizer. Hints are a powerful tool that can be used to analyze and explore alternative query plans and to work around problems. Nevertheless, hints should be used with caution. More often than not, the optimizer does a fine job and selects the best (or, at the very least, a perfectly reasonable) query plan. Before deciding to use hints, keep the following tips in mind:

- Make sure there is, indeed, a performance problem. Just because the performance of a query does not meet expectations does not imply that a better plan exists. Often the query plan selected by the optimizer may, indeed, be the best query plan. The "problem" query may simply be a big or complex query and using hints improperly may yield a worse plan.
- Keep in mind that not all performance issues are caused by poor plan choices. In some cases, there may be other issues (for example, locking and concurrency problems or memory contention) that are leading to a performance problem. It is well worth exploring and eliminating these potential issues before resorting to hints.
- Even if you have a plan issue, the problem might be resolved by updating statistics (or by collecting statistics with a bigger sample) or by a schema change. For example, creating a new index or constraint might enable the optimizer to choose a better plan. It is generally worth exploring and understanding the underlying cause of a poor plan choice before trying to force a better plan using hints.
- Improperly used hints can cause the query optimizer to fail to find a plan. Hints can only direct the optimizer to select a plan that it would explore or consider even without the use of hints. Hints cannot make the optimizer choose a plan (not even a valid plan) that it would not explore or consider in the absence of the hint. This limitation is caused by the internal mechanisms that the optimizer uses to implement hints, but is also critical for ensuring that hints are not used to generate an invalid plan that might result in incorrect results or corrupted data.

Tip



Hints can in some cases cause the optimizer to change the set of plans that it explores. For example, in general, the optimizer does not consider bushy join trees. (See [Chapter 3](#) for an explanation of the difference between left deep, right deep, and bushy join trees.) However, certain hints (for example, FORCE ORDER and USE PLAN) can cause the optimizer to consider these plan alternatives.

- If you do choose to use hints in an application, try to use them as little as possible. Keep in mind that hints are usually chosen to achieve a desired result against one version of SQL Server with one set of data. The supplied hint may or may not achieve a satisfactory result if the data changes due to many updates or new rows added to the tables and a hint may even cause the optimizer to fail to find a plan against a future version of SQL Server. Consider using plan guides (new to SQL Server 2005) to separate the specification of the hints from the application itself. Using plan guides will make it easier to update or remove the hints later on if they prove inappropriate without the need to update the application. Plan guides are discussed in more detail in [Chapter 5](#).

Before trying to use hints to diagnose or solve a problem, it helps to narrow down the scope of the problem by simplifying the query and/or by identifying which part of the query plan is the problem. Here are some things to try:

- If you have a complex insert, update, or delete query, try rewriting it as a select query to see whether the problem is related to the insert, update, or delete. For queries that return many rows, you might need to use a SELECT INTO or an aggregation [for example, a COUNT(*)] to eliminate the overhead of returning these rows to the client.
- Try removing joins, subqueries, or aggregations from the query and see whether you can still observe the performance problem.
- Focus on those operators that the optimizer estimates have the highest cost. Use STATISTICS PROFILE or STATISTICS XML output to look for operators that are processing many rows or nested loops joins with many executions. Try to verify whether the optimizer cost estimates are valid and investigate any serious errors in the cardinality estimates.

Once you have identified the potential source of a problem, you can use hints to explore alternate query plans and try out solutions or workarounds. In this section, we look only at hints that affect plan selection. SQL Server also supports hints that affect locking, plan caching, and recompilation. Plan caching and recompilation hints are covered in [Chapter 5](#). Locking hints are covered in [Chapter 6](#), "Concurrency Problems."

Plan selection hints can be broken down into three categories based on the portion of the query plan that the hint impacts. These categories are: query hints, table hints, and join hints. Query hints affect the entire query, table hints affect a single table referenced by the query, and join hints affect a single join.

Plan selection hints can also be categorized into two groups that we might call goal-oriented and physical operator hints. A goal-oriented hint conveys a logical goal to the optimizer, without specifying how the optimizer should achieve that goal, which physical operators the optimizer should use, or how to arrange those operators. The FAST N and OPTIMIZE FOR hint, which we will discuss later in this chapter, are examples of goal-oriented hints. A physical operator hint, on the other hand, tells the optimizer to use a specific operator or set of operators in the query plan or to organize the operators in the query in a particular way without giving any indication of what behavior we are trying to achieve or why we want the particular plan that we are forcing. Join hints, FORCE ORDER, and USE PLAN are all examples of physical operator hints.

Goal-oriented hints have some advantages over physical operator hints:

- They often can be used effectively even without deep knowledge or understanding about the query plan or the reasons behind why it does not perform as well as you would like. It is much more difficult to use a physical hint without a solid understanding of the query plan you are trying to improve.
- All hints carry some risks of unintended consequences. However, because goal-oriented hints only specify a goal without specifying how to achieve the goal, they are frequently safer than physical operator hints. You can use a goal-oriented hint without worrying about causing the optimizer to fail to generate a plan. A goal-oriented hint also is more likely to continue to work as expected and have the desired effect after schema changes or server upgrades.

Physical operator hints, however, are more useful when you have in mind a specific plan or a specific change to a plan. It can be difficult or even impossible to force a specific plan using a goal-oriented hint.

Query Hints

There are many query hints available, including hints to force the join type, aggregation type, union type, the join and aggregation order, and more. Because query hints affect the entire query, if a query includes more than one join, aggregation, or union operation, a join, aggregation, or union hint will affect all of the joins,

aggregations, or unions in the query. It is not possible with a query hint to limit the hint to a specific operator or portion of the plan. (But keep in mind that SQL Server does have join hints to affect a single join. We will discuss join hints separately.)

Query hints are specified in the OPTION clause, which can only be used at the end of a query and only on the outermost or main query. Query hints may not be specified on subqueries and may not be used in Common Table Expression (CTE), view, or inline TVF definitions. However, any query hints on the main query do affect all subqueries, CTEs, views, and inline TVFs. In fact, since the definitions of many views and inline TVFs are complex queries, when queries contain views or inline TVFs, query hints should be used with caution to ensure that they do not have any unexpected or unintended consequences on the portions of the query plan corresponding to the views or inline TVFs.

FAST N Hint

The FAST N hint is a goal-oriented hint. It tells the optimizer to try to choose a plan that will produce the first N output rows as quickly as possible. This hint may be useful for applications that must display a screenful of data quickly. In general (and especially for small values of N), this hint discourages the optimizer from choosing a plan with blocking operators that might delay output of the first results. For example, for small values of N, the FAST N hint will typically discourage the optimizer from using hash join, hash aggregation, sorts, and even parallelism. Note that most blocking operators also happen to be memory consumers, so while there are no guarantees, specifying a FAST N hint may also be useful for generating a query plan that minimizes or discourages memory use.

The benefits of the FAST N hint do not come for free. The price for quickly generating the first N results may be delays— in some cases significant delays—in the overall response time of the query. In other words, while the FAST N hint may cause the optimizer to pick a plan that generates the first N output rows quickly, it may also cause the optimizer to generate a plan that takes much longer, consumes many more CPU cycles, and performs many more I/Os before it finishes producing the last row.

The FAST N hint has a similar effect to a TOP N clause with respect to query optimization and plan selection. The principle difference between FAST N and TOP N is, of course, that FAST N only optimizes as if we have a TOP N clause but produces a plan that will ultimately output all of the rows in the original query, whereas a TOP N clause actually limits the number of rows that the query generates to N.

The FAST N hint makes a best effort, but will never cause optimization to fail (any more than TOP N will cause a query to fail). For example, the optimizer might try if possible to substitute an index in place of an explicit sort or to substitute a stream aggregate in place of a hash aggregate, but if there are no valid alternative plans, the optimizer will generate a plan with whatever operators it must use regardless of whether or not those operators are blocking.

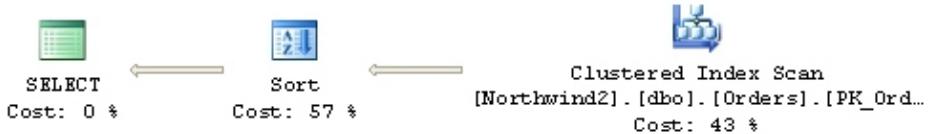
The value of N matters with a FAST N hint. FAST 1 and FAST 100 are not equivalent and may result in very different plans. At one extreme, if the optimizer expects a query to produce fewer than N rows, it simply discards the hint entirely. In that case, the hint has no effect on the optimization process or on the resulting query plan. Even if the optimizer expects the query to produce more than N rows, as we increase N, we increase the likelihood that the optimizer will decide that a plan with blocking operators is more efficient given the number of rows we are requesting, and it will choose a plan with these blocking operators.

Let's look at an example. Consider the following query:

```
SELECT [OrderId], [CustomerId], [OrderDate]
FROM [Orders]
ORDER BY [OrderDate]
```

This query requests the entire Orders table sorted on the OrderDate column. Although we have a nonclustered index on OrderDate, this index does not cover the CustomerId column. Thus, the optimizer must choose between a clustered index scan with a sort and a nonclustered index seek with a bookmark lookup, and since there is no WHERE clause, SQL Server will have to do a bookmark lookup for every row in the table. This is a classic scenario and in this case, the optimizer chooses the scan and sort with its sequential I/Os over the seek and bookmark lookups with its random I/Os, as shown in [Figure 4-5](#):

Figure 4-5. Query plan showing clustered index scan and sort



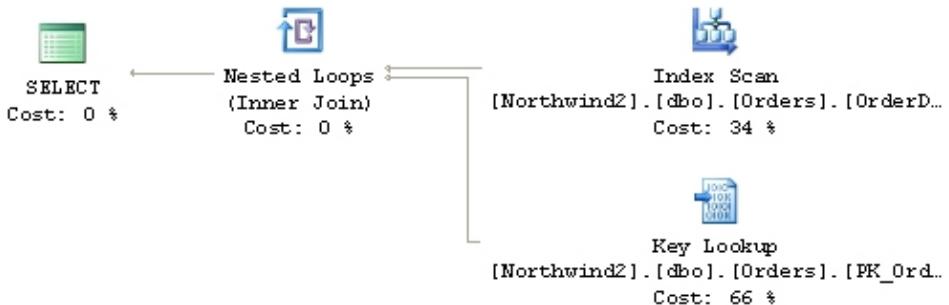
Now let's add a FAST 1 hint to the same query:

```

SELECT [OrderId], [CustomerId], [OrderDate]
FROM [Orders]
ORDER BY [OrderDate]
    
```

With the hint, the optimizer determines that performing one random I/O to return one row is faster than scanning and sorting the entire table. The graphical plan for this query is shown in [Figure 4-6](#), followed by the text plan:

Figure 4-6. Query plan using FAST hint, showing scan of nonclustered index and clustered index seek



Code View:

```

--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID], [Expr1003]))
WITH ORDERED PREFETCH
|--Index Scan(OBJECT:([Orders].[OrderDate])), ORDERED FORWARD
|--Clustered Index Seek(OBJECT:([Orders].[PK_Orders])),
SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD
    
```

Now suppose we change the FAST 1 hint to a FAST 100 hint:

```
SELECT [OrderId], [CustomerId], [OrderDate]
FROM [Orders]
ORDER BY [OrderDate]
```

Although the Orders table has about 800 rows, the optimizer concludes that the sequential scan is still less expensive than 100 random I/Os and returns to the original plan with the sort. In fact, the cost of random I/Os is so high, that for this example, the crossover point where the optimizer switches back to sequential I/Os occurs with a FAST 16 hint. Note that 16 rows are only 2 percent of the 800 total rows in the table! While this threshold may seem surprisingly low, this behavior is exactly the same as what we observe any other time the optimizer must choose between performing a sequential scan of a base table or a nonclustered index seek with a bookmark lookup.

OPTIMIZE FOR Hint

The OPTIMIZE FOR hint is another goal-oriented hint, new in SQL Server 2005. It asks the query optimizer to generate a plan based on the parameter values that you specify. This hint can be especially useful with a skewed dataset, in which different parameter values may result in much different cardinality estimates and different plans. If one of the plans is more robust than others or if some parameter values are more commonly used during execution than others, this hint offers a simple way to get the desired or best plan without the need to resort to physical operator hints, to embed constants into your queries using dynamic SQL (which has a multitude of drawbacks), or to force frequent and possibly expensive recompilations. This hint can be useful with queries that are submitted directly in a batch and which cannot benefit from parameter sniffing. It can be equally useful with stored procedures where parameter sniffing sometimes leads to poor plans depending on which parameter values were used with the stored procedure when it was compiled.

Like the FAST N hint, the OPTIMIZE FOR hint will never cause optimization to fail. However, use caution when choosing the sample parameter values to be used with the OPTIMIZE FOR hint. The parameter values that you select must remain representative of the case for which you are trying to optimize over a potentially long period of time— perhaps even over the lifetime of your application. If over time, perhaps because of ongoing inserts, updates, or deletes to your data, the values you choose cease to be good choices, you may end up forcing a less-than-optimal query plan. One potential solution to this problem that may help in some cases is to add a dummy value to your dataset that is never modified but is, by design, representative of a good parameter value.

Let's look at an example of how this hint works. In the Northwind2 database, the number of orders shipped to different postal codes varies from only 1 order to postal code "05022" up to 31 orders to postal code "83720." Consider the following query to look up orders shipped to a specific postal code:

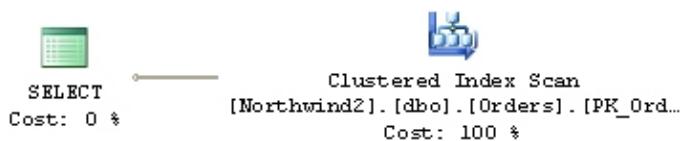
```
SELECT [OrderId], [OrderDate]
FROM [Orders]
WHERE [ShipPostalCode] = N'05022'
```

For a postal code with few orders (such as 05022), the optimizer may choose an index seek of the ShipPostalCode nonclustered index followed by a bookmark lookup. For a postal code with many orders, the optimizer may instead choose a clustered index scan. Now let's observe what happens if we use a variable instead of a constant:

```
DECLARE @ nvarchar(20)
SET @ = N'05022'
SELECT [OrderId], [OrderDate]
FROM [Orders]
WHERE [ShipPostalCode] = @
```

Although this batch includes an actual value for the @ShipCode variable, the optimizer compiles the SELECT statement before it executes the assignment SET statement, so at optimization time the value of the variable @ShipPostalCode is unknown. Without an actual value for the variable, the optimizer is unable to generate an accurate cardinality estimate for the predicate. Forced to choose from a range of possible cardinalities (any of which would be valid for some parameter value), the optimizer chooses an answer somewhere in the middle. If we check the query plan, we see that the optimizer chooses a cardinality estimate of nearly 10 rows. This estimate is sufficiently high that the optimizer chooses the clustered index scan plan, as shown in [Figure 4-7](#):

Figure 4-7. Query plan showing clustered index scan when the value being searched for is unknown to the optimizer



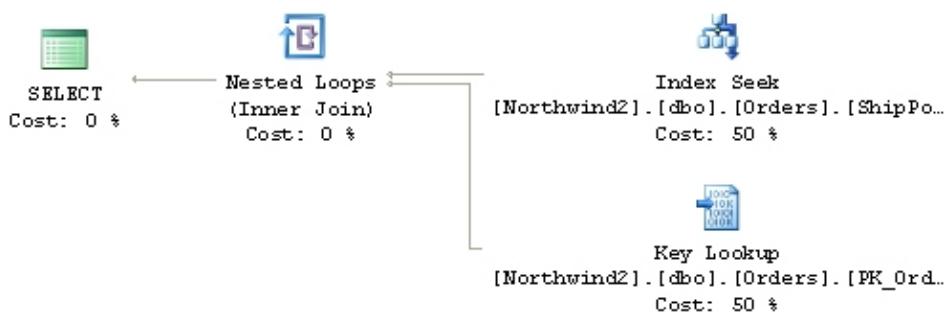
Now let's see what happens when we add an OPTIMIZE FOR hint to the query:

```

DECLARE @ nvarchar(20)
SET @ = N'05022'
SELECT [OrderId], [OrderDate]
FROM [Orders]
WHERE [ShipPostalCode] = @
  
```

This time the optimizer generates an accurate cardinality estimate of just one row and chooses the plan with the nonclustered index seek and bookmark lookup. The plan is shown in [Figure 4-8](#):

Figure 4-8. Query plan showing that a different plan can be chosen when the OPTIMIZE FOR hint is used



Note that it does not really matter what the actual parameter value is. As long as we use the OPTIMIZE FOR hint and tell the optimizer to use postal code 05022, we will get the same plan regardless of whether we actually search for postal code 05022, 83720, or anything in between. We can verify that the correct parameter value was used during compilation by looking at the Properties window along a graphical query plan, or by examining the SHOWPLAN_XML output:

```

<QueryPlan ...>
  <ParameterList>
  
```

```
</ParameterList>
</QueryPlan>
```

Of course, in this example we could also get the same plan by forcing a recompile using the RECOMPILE query hint (discussed in [Chapter 5](#)). The principle difference is that the OPTIMIZE FOR hint only requires a single compilation (and only takes effect when the query is compiled), whereas the RECOMPILE hint would force a recompilation on every execution.

Query-Level Join Hints

Now that we've looked at some goal-oriented hints and seen how they work and how we can use them, let's take a look at some physical operator hints. Let's begin with query-level join hints. These hints are used to force the optimizer to choose a specific join type. There are three query-level join hints— one for each physical join operator. The hints are LOOP JOIN, MERGE JOIN, and HASH JOIN. If you specify one of these hints, it forces all joins in the query plan to use that join type. If you specify two of these hints, it forces all joins in the query plan to use one of the two specified join types. In other words, if you specify two of these hints, it has the effect of disabling the third unspecified join type. The optimizer makes a cost-based decision as to which of the remaining join types to use for each join.

Using two query-level join hints can be especially useful if the query plan chosen by the optimizer includes one undesirable join. For example, suppose that we want a query plan that does not require a memory grant and the optimizer chooses a query plan with a hash join. We can specify OPTION (LOOP JOIN, MERGE JOIN) to eliminate the hash join. Of course, this hint does not guarantee that the optimizer will not add another memory-consuming operator such as a sort.

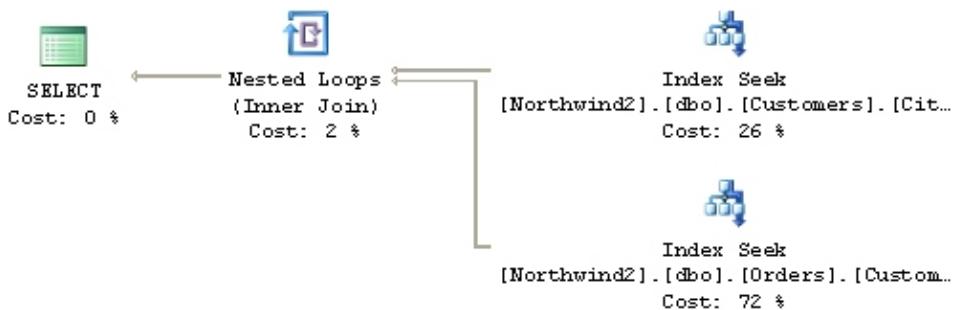
Keep in mind that by changing a join type, you may also cause the optimizer to change many other aspects of the query plan. The optimizer tries to find the best plan using the specified join type(s) and the best plan with one join type may differ significantly from the best plan with another join type. The optimizer may choose a different join order, it may choose different indexes, it may change seeks into scans or vice versa, and so on. For instance, if you force a nested loops join, the optimizer may choose an index seek based on a correlated parameter for the table on the inner side of the join. Or, if you force a merge join, the optimizer may choose indexes that provide the necessary sort order for the merge join. If no appropriate index is available or if an index is available but the cost of a plan with that index is too high, the optimizer may even introduce one or more sort operators into the query plan.

To see how we can use query-level join hints, consider the following query, which looks for orders placed by customers living in London:

```
SELECT O.[OrderId]
FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London'
```

The query optimizer chooses a nested loops join for this query, as shown in [Figure 4-9](#).

Figure 4-9. Query plan showing a nested loops join



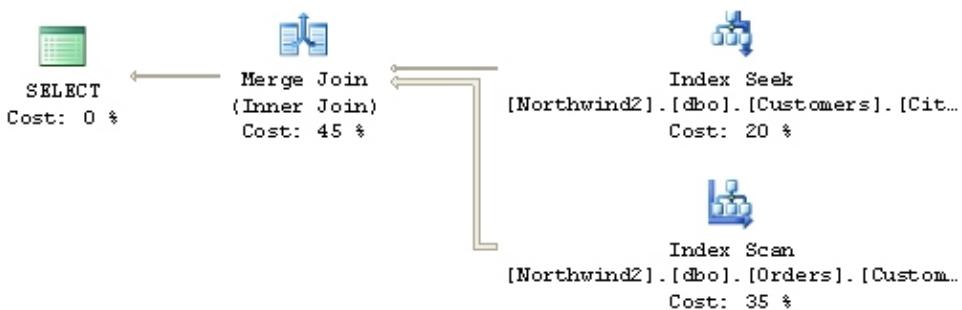
If we want to force a merge join, we simply need to add a MERGE JOIN hint to the query as follows:

```

SELECT O.[OrderId]
FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London'
  
```

The new query plan uses a merge join like we requested. The plan is shown in [Figure 4-10](#).

Figure 4-10. Query plan that has been hinted to use a merge join



Notice that the index seek on the Orders table became an index scan with the merge join plan. Without the nested loops join, there is no correlated parameter to use to perform an index seek.

We can use query-level join hints to force the join type for explicit joins that are part of a query as well as for most implicit joins introduced by the optimizer. For example, we can force the join type used by index intersections, although forcing the join type for an index intersection is rarely, if ever, a good idea, as the optimizer nearly always makes an appropriate choice. We can even force the join type used for foreign key validation, cascading actions, and index view maintenance. There are only a few places where SQL Server 2005 uses joins but where we cannot force the join type. These special cases include bookmark lookups, dynamic index seeks, and the joins used to enumerate partition ids for partitioned tables. These joins ignore any query-level join hints and always use nested loops joins. It would not make much sense for a bookmark lookup to use a merge or hash join. For example, recall that a bookmark lookup takes a set of clustering keys or record ids and looks up just these keys or record ids in the base table. Using a merge or hash join for a bookmark lookup would require that SQL Server perform a scan of the entire clustered index or heap, which would defeat the point of the bookmark lookup. If the server is going to scan the entire base table, it does not need the bookmark lookup in the first place.

Caution

 Be careful when using query-level join hints. It must be possible to perform all joins in the query using the forced join type(s). For example, using a HASH JOIN hint on a query that includes a non-equijoin prevents the query optimizer from finding a valid plan and results in an error.

Tip

 The query-level join hints are blunt instruments. It is not possible to force different join types for different joins in a query plan. For simple queries with only one or two joins or for queries with an implicit join (which cannot be forced by any other means), a query join hint may be appropriate. However, if you have a nontrivial query with many joins and if you wish or need to control the join type of each join independently, consider using the ANSI-style join hints described later in this chapter. ANSI-style join hints afford much more precise join by join control.

GROUP Hints

Just as we can force join types with join hints, we can also force aggregation types with GROUP hints. Moreover, just as there are three join hints— one for each physical join operator— there are two group hints, again one for each physical aggregation operator. The group hints are ORDER GROUP and HASH GROUP. ORDER GROUP forces the query optimizer to choose stream aggregation (recall that the stream aggregate operator requires that input data be sorted on the group by keys, hence the keyword ORDER), whereas HASH GROUP forces the query optimizer to choose hash aggregation. Again, just as a query-level join hint forces all joins in the query to use the specified join type, a group hint affects all aggregations performed by the query plan. If you have a query plan with multiple aggregation operators (for example, because of aggregations performed in subqueries or views), using one of these hints forces all of the aggregation operators to use the specified physical aggregation operator. There is no way with a GROUP HINT to force the optimizer to use a stream aggregate for some aggregation operations and a hash aggregate for others.

Note

 It is possible to force a mix of stream and hash aggregates in a single plan with the USE PLAN hint discussed later in this chapter.

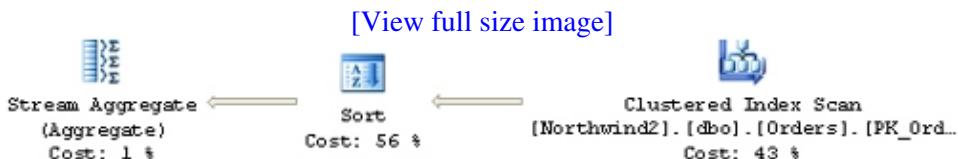
Since the stream aggregate operator requires data to be sorted on the GROUP BY keys, using the ORDER GROUP hint may encourage the optimizer to choose appropriate indexes and/or join types to achieve the necessary order without sorting. If the optimizer cannot find a plan that provides the necessary order without sorting or if it decides that such a plan is more expensive than sorting, it may introduce a sort operator. This effect is similar to how a MERGE JOIN hint can influence the optimizer to choose a different index selection or introduce sorts.

For an example of how we can use a GROUP hint, consider this query, which computes the most recent order date for each customer:

```
SELECT [CustomerId], MAX([OrderDate])
FROM [Orders]
GROUP BY [CustomerId]
```

The optimizer chooses a plan with a sort and stream aggregate, as shown in Figure 4-11.

Figure 4-11. Query plan showing sort and stream aggregation



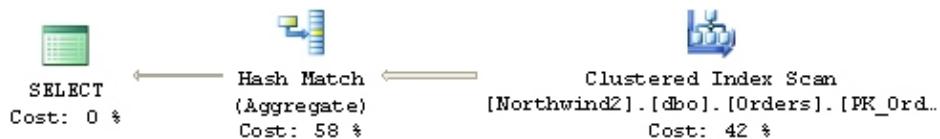
We can easily force a hash aggregate with the HASH GROUP hint:

```

SELECT [CustomerId], MAX([OrderDate])
FROM [Orders]
GROUP BY [CustomerId]
    
```

Notice that the resulting query plan does not need a sort, as shown in Figure 4-12.

Figure 4-12. A query plan hinted to use hash aggregation



Caution



SQL Server always uses the stream aggregate operator for scalar aggregations. There is no way to force the optimizer to use a hash aggregate for a scalar aggregation. If you attempt to use the HASH GROUP hint on a query with a scalar aggregation, the optimizer will fail with an error.

UNION Hints

In addition to forcing join and aggregation types, we can also force one of the three possible union operators. The three UNION hints are CONCAT UNION, MERGE UNION, and HASH UNION. The MERGE UNION and HASH UNION hints are fairly straightforward and merely force the optimizer to choose a merge union or hash union operator, respectively. The CONCAT UNION hint is slightly trickier. It forces the optimizer to choose either a regular (unordered) concatenation operator or an order-preserving merge concatenation operator. Since these concatenation operators do not remove duplicates, if you use the CONCAT UNION hint, the optimizer must add another operator (for example, a distinct sort or an aggregation operator) to remove the duplicates.

Note



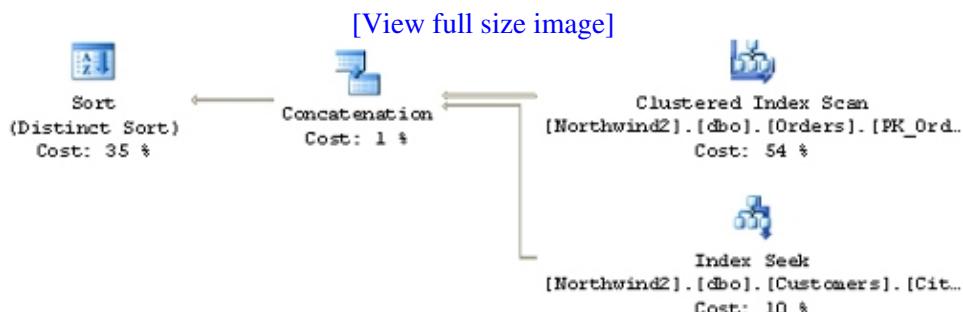
These hints are only interesting for duplicate eliminating UNION queries. UNION ALL queries do not remove duplicates and, thus, are always implemented using either the regular unordered concatenation operator or an order-preserving merge concatenation operator.

To see an example of how we can use this hint, consider the following query, which returns a list of customers who have had orders shipped to London or who live in London:

```
SELECT [CustomerId]
FROM [Orders]
WHERE [ShipCity] = N'London'
UNION
SELECT [CustomerId]
FROM [Customers]
WHERE [City] = N'London'
```

Without any hints, the optimizer chooses a plan with a regular concatenation operator followed by a sort distinct, as shown in [Figure 4-13](#).

Figure 4-13. Query plan for a UNION, processed as concatenation followed by a sort



Now observe what happens if we add a MERGE UNION hint:

```
SELECT [CustomerId]
FROM [Orders]
WHERE [ShipCity] = N'London'
UNION
SELECT [CustomerId]
FROM [Customers]
WHERE [City] = N'London'
```

As instructed, the optimizer chooses a plan with a MERGE UNION, as shown in [Figure 4-14](#).

Figure 4-14. Query plan for a UNION operation, with a forced MERGE UNION hint

[\[View full size image\]](#)



The final sort is gone. We no longer need it since the merge union eliminates duplicates. However, the merge union requires both of its inputs to be sorted by CustomerId and for duplicate CustomerId values to be eliminated from the individual inputs. The index seek on the Customers table is already sorted and the CustomerId column is a primary key, so there are no duplicate values to eliminate. The clustered index scan of the Orders table is not sorted by the CustomerId column and may contain duplicates. Thus, the optimizer adds a new sort distinct between the scan of the Orders table and the merge union.

Force Order

The FORCE ORDER hint allows us to control the join order and aggregation placement selected by the optimizer.

Forcing Join Order

We'll begin by discussing how we can use this hint to force the join order. The FORCE ORDER hint tells the optimizer to join tables in the same order that they appear in the FROM clause of the query. If the FROM clause consists purely of a comma-separated list of base tables, the FORCE ORDER hint results in a left deep tree where the tables are joined in the order that they appear in the FROM clause.

We can force right deep or bushy trees by using the ANSI join syntax (in other words, the JOIN keyword) or by using FROM clause subqueries. To force right deep or bushy trees using the ANSI join syntax, we just need to use parentheses to group the tables in the order that we want them to join. If we join three or more tables using the ANSI syntax and omit the parentheses, the placement of the ON clauses determines in which order we join the tables and determines whether we get a left deep or a right deep tree. For example, the following two queries are semantically equivalent, but using the FORCE ORDER and HASH JOIN hints and moving the ON clause that links Employees and Orders changes the join order. The join appearing in bold is processed first.

```

SELECT O.[OrderId]
FROM [Customers] C JOIN
     ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London' AND E.[City] = N'London'
OPTION (FORCE ORDER, HASH JOIN)

```

```

SELECT O.[OrderId]
FROM
     JOIN [Employees] E ON O.[EmployeeId] = E.[EmployeeId]
WHERE C.[City] = N'London' AND E.[City] = N'London'
OPTION (FORCE ORDER, HASH JOIN)

```

Forcing right deep or bushy trees using FROM clause subqueries is similar. Select a group of tables that forms a left deep subtree and join this group of tables using a subquery. Be sure to list the tables in the FROM clause in the order that you want them to join. Now treat this subquery as if it were a regular table (it might help to think of it as a view), select another group of tables that forms another left deep subtree, and repeat the process. Continue adding more tables until all of the tables are included in the final query. As you build up each FROM clause, list base tables and subqueries in the order that you want them to join.

Note also that if the FROM clause contains CTEs, views, or inline TVFs, these constructs behave just like subqueries as far as the FORCE ORDER hint is concerned. Note also, if we have a query with outer joins, we may not be able to force all possible join orders without changing the semantics of the query.

Caution



The FORCE ORDER hint is extremely powerful; use it with caution. Changing the join order of a query may have many unintended, unexpected, and/or undesirable side effects. The optimizer may choose to use different indexes or join types, it may choose a scan instead of a seek (for instance if the seek depended on a correlated parameter, which is no longer available because of the new join order) or a seek instead of a scan, it may introduce or eliminate bookmark lookups, and so on.

Of particular concern, with the FORCE ORDER hint it is not hard to introduce a cross join accidentally with catastrophic results on performance. Unless it is your intention to use a cross join, be sure that each table after the first one in the FROM clause directly joins with at least one table previously referenced in the FROM clause. This risk is even more of a concern for queries that reference views and/or inline TVFs. Although everything may appear reasonable in the main query, the FORCE ORDER hint also applies to the FROM clauses in the views and/or inline TVFs where, without realizing it, the hint might introduce a cross join.

The ANSI join syntax significantly reduces the risk of inadvertently introducing cross joins. You must specify an ON clause for all inner and outer joins. If you accidentally omit the ON clause, SQL Server generates a syntax error. Thus, it is more difficult to force a cross join accidentally when using the ANSI join syntax with the FORCE ORDER hint. (Even with the ANSI join syntax it is still possible to write a join predicate that will yield a cross join. For example, you can write a join predicate that only references tables from one side of the join.)

The FORCE ORDER hint can also cause unintended side effects for queries with non-FROM clause subqueries. Normally, the optimizer explores join orders involving all of the tables in a query plan, including those tables that appear in subqueries regardless of whether those subqueries appear in the FROM clause, the SELECT list, or the ON, WHERE, or HAVING clauses of a query. The FORCE ORDER hint forces the "local" join order for tables listed in the FROM clause of each subquery regardless of where in the main query it appears. It also prevents the optimizer from considering alternative "global" join orders across all of the tables in all of the subqueries within a single query. It is essentially impossible to control the overall join order for a query with non-FROM clause subqueries.

Choosing an Appropriate Join Order

Choosing an appropriate join order is a tricky problem. It is a key element of any query optimizer and is the subject of much research and many technical papers. To get some idea of how complex this problem really is, one just needs to consider the number of possible join orders for queries with increasing numbers of tables and joins: [Table 4-2](#) indicates the number of join orders of different shapes that are possible with a given number of tables in the query.

Table 4-2. Number of Possible Join Orders for a Given Number of Tables in a Query

# of Tables	# of Left Deep Join Orders	# of Bushy Tree Join Orders
2	2	2
3	6	12
4	24	44
5	120	205
6	720	1680
7	4320	6720
8	25,920	30,240

As you can see, the number of possible join orders increases exponentially with the number of tables and becomes quite large even for relatively small numbers of tables. Moreover, this analysis considers the number of join orders only; it does not account for other query plan alternatives such as join type, placement of an aggregation operator, or index choices. We must consider all of these factors to determine the plan cost. So, given the complexity of this problem, how should you go about choosing an appropriate join order when you use the FORCE ORDER hint? Here are a few tips:

- Start with the plan selected by the optimizer. For complex queries with many joins, there is a good chance that the plan selected by the optimizer is fairly reasonable. If there is one join or one portion of the plan that you believe is faulty, try to force the same plan with the minimum necessary changes to resolve the problem. Note that it is not always possible or it may be extremely difficult to force a specific plan or a specific join order. For instance, it may be difficult to force a particular join order for a query that involves non-FROM clause subqueries.
- If you are upgrading and encounter a problem with the plan selected by the new version of SQL Server, you may want to start with the plan selected by a prior version of the server. However, keep in mind that in some cases the new optimizer may no longer support the old plan and that we cannot use hints to force a query plan that the optimizer cannot generate.
- Try to follow the natural join order and to take advantage of foreign key/primary key relationships. Avoid cross joins or Cartesian products except in rare cases in which the cardinality of one or both of the inputs is very small. For example, a cross join might be reasonable if you know that one of the inputs produces only one row.
- Try to begin by joining small tables or tables that will filter out as many rows as possible. Try to delay joining larger tables or tables that will increase the cardinality of the result set. For example, if you have a star schema, join the dimension tables with the most selective filters before joining with a large fact table.
- In general, it helps to push aggregations that significantly reduce cardinality as early in the query plan as possible. However, bear in mind that it may still be better to push a highly selective join before an aggregation. We'll discuss forcing aggregation placement later in this section.
- With nested loops joins, try to join small tables on the outer side of the join and try to choose tables with appropriate indexes on the inner side.
- With hash joins, try to place the smaller table or input on the build side of the join and the larger table or input on the probe side of the join. This will minimize the memory required by the hash join and will minimize the risk that the join runs out of memory and spills rows to disk.

FORCE ORDER Example

Let's look at a simple example of how we can use the FORCE ORDER hint to change the join order. For this example, consider the following query which joins three tables:

```
SELECT O.[OrderId]
FROM [Employees] E JOIN [Orders] O
ON O.[EmployeeId] = E.[EmployeeId]
```

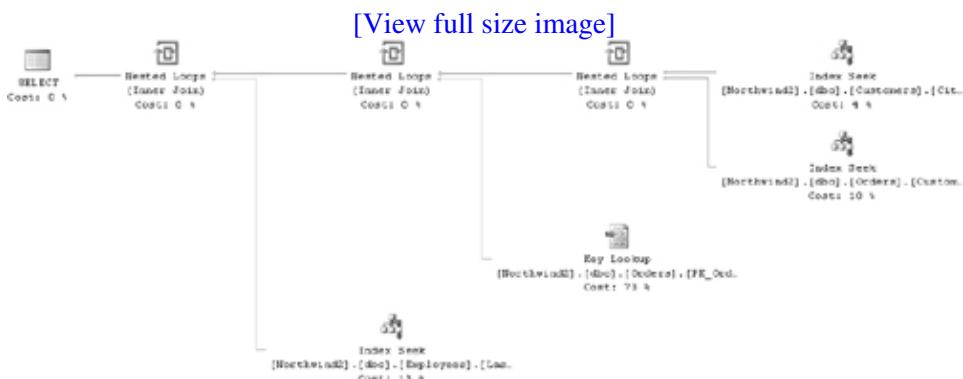
```

JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]
WHERE
E.[LastName] = N'Peacock' AND
C.[City] = N'London'

```

The optimizer chooses the following left deep plan with the left deep join order Customers to Orders to Employees. The graphical plan is shown in [Figure 4-15](#) and is followed by the text plan for comparison.

[Figure 4-15. Query plan for a left deep join](#)



Code View:

```

|--Nested Loops(Inner Join, OUTER REFERENCES:([O].[EmployeeID]))
 |--Nested Loops(Inner Join, OUTER REFERENCES:([O].[OrderID], [Expr1006])
    WITH UNORDERED PREFETCH)
 |   |--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
 |   |   |--Index Seek(OBJECT:([Customers].[City] AS [C]),
 |   |   |   SEEK:([C].[City]=N'London') ORDERED FORWARD)
 |   |   |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
 |   |   |   SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
 |   |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders] AS [O]),
 |   |   SEEK:([O].[OrderID]=[O].[OrderID]) LOOKUP ORDERED FORWARD)
 |--Index Seek(OBJECT:([Employees].[LastName] AS [E]), SEEK:([E].
 |   |   [LastName]=N'Peacock' AND [E].[EmployeeID]=[O].[EmployeeID]) ORDERED FORWARD)

```

Now observe how the join order changes if we include the FORCE ORDER hint:

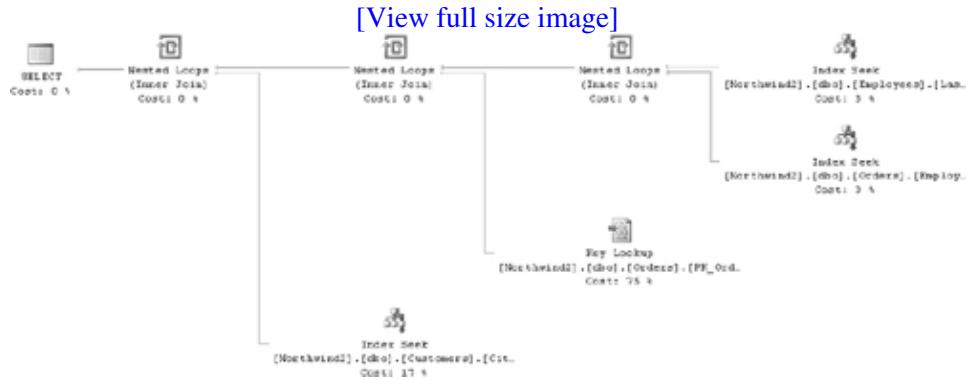
```

SELECT O.[OrderId]
FROM [Employees] E JOIN [Orders] O
ON O.[EmployeeId] = E.[EmployeeId]
JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]
WHERE
E.[LastName] = N'Peacock' AND
C.[City] = N'London'

```

With the FORCE ORDER hint, we still get a left deep plan, but now the join order matches the FROM clause order of Employees to Orders to Customers, as you can see in the graphical plan in [Figure 4-16](#). The text plan follows to allow you to see the join order listed sequentially.

Figure 4-16. Query plan for a left deep join with a forced join order



Code View:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([O].[CustomerID]))
 |--Nested Loops(Inner Join, OUTER REFERENCES:([O].[OrderID], [Expr1006])
    WITH UNORDERED PREFETCH)
 |   |--Nested Loops(Inner Join, OUTER REFERENCES:([E].[EmployeeID]))
 |   |   |--Index Seek(OBJECT:([Employees].[LastName] AS [E]),
 |   |   |   SEEK:([E].[LastName]=N'Peacock') ORDERED FORWARD)
 |   |   |--Index Seek(OBJECT:([Orders].[EmployeeID] AS [O]),
 |   |   |   SEEK:([O].[EmployeeID]=[E].[EmployeeID]) ORDERED FORWARD)
 |   |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders] AS [O]),
 |   |   SEEK:([O].[OrderID]=[O].[OrderID]) LOOKUP ORDERED FORWARD)
 |--Index Seek(OBJECT:([Customers].[City] AS [C]), SEEK:([C].[City]=N'London'
 AND [C].[CustomerID]=[O].[CustomerID]) ORDERED FORWARD)
```

Next, suppose that we wish to force the right deep join order Employees to Customers to Orders, where Orders is the rightmost table in the join. We can force this plan either by grouping the joins using parenthesis or by using a subquery. The ANSI join alternative is generally a bit simpler and may appear more natural:

```
SELECT O.[OrderId]
FROM [Employees] E JOIN
(
    [Customers] C JOIN [Orders] O
    ON O.[CustomerId] = C.[CustomerId]
)
ON
O.[EmployeeId] = E.[EmployeeId]
WHERE
C.[City] = N'London' AND
E.[LastName] = N'Peacock'
OPTION (FORCE ORDER)
```

Here is the same query rewritten with a subquery:

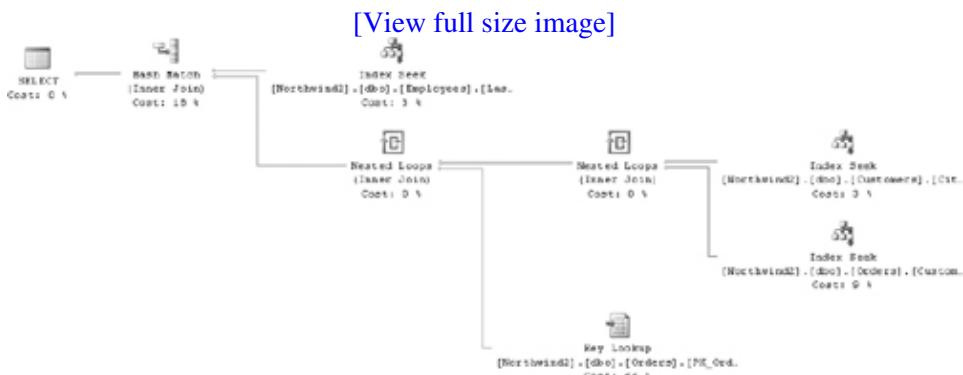
```

SELECT CO.[OrderId]
FROM [Employees] E,
(
    SELECT O.[OrderId], O.[EmployeeId]
    FROM [Customers] C, [Orders] O
    WHERE
        O.[CustomerId] = C.[CustomerId] AND C.[City] = N'London'
)
CO
WHERE
    CO.[EmployeeId] = E.[EmployeeId] AND E.[LastName] = N'Peacock'
OPTION (FORCE ORDER)

```

The optimizer honors the FORCE ORDER hint by joining Employees to the result of the Customers and Orders join. Because the Employees table appears first in the FROM list in both variations of the query, the optimizer guarantees that the Employees table is the left input and the result of the Customers and Orders join is the right input to the second join. The resulting plan is the same for both versions of the query and can be seen in the graphical plan in [Figure 4-17](#) and in the text plan that follows.

Figure 4-17. Query plan for a right deep join with a forced join order



Code View:

```

|--Hash Match(Inner Join, HASH:([E].[EmployeeID])=([O].[EmployeeID]), RESIDUAL:(...))
|--Index Seek(OBJECT:([Employees].[LastName] AS [E]),
  SEEK:([E].[LastName]=N'Peacock') ORDERED FORWARD)
|--Nested Loops(Inner Join, OUTER REFERENCES:([O].[OrderID], [Expr1006])
  WITH UNORDERED PREFETCH)
    |--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
      |--Index Seek(OBJECT:([Customers].[City] AS [C]),
        SEEK:([C].[City]=N'London') ORDERED FORWARD)
      |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
        SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders] AS [O]),
      SEEK:([O].[OrderID]=[O].[OrderID]) LOOKUP ORDERED FORWARD)

```

Note that with the new join order, the costing has changed and one of the joins that was previously a nested loops join is now a hash join.

Forcing Aggregation Placement

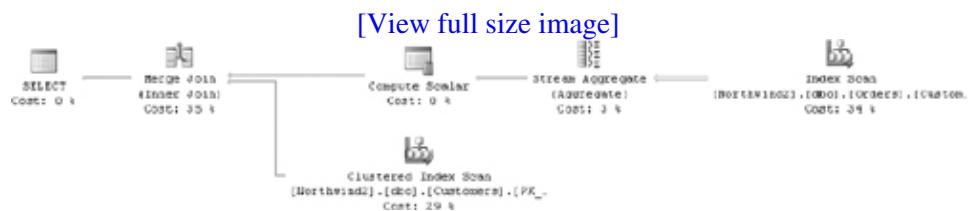
Now that we've seen the effect of the FORCE ORDER hint on join order, let's look at how we can use this hint to affect aggregation placement. Normally (in the absence of any hints), the optimizer may choose to move an aggregation earlier or later in the query plan based on costing. However, if we use the FORCE ORDER hint on a query that includes a subquery with a GROUP BY clause, the optimizer joins the tables in the subquery first, performs the aggregation, and then continues joining any remaining tables. The optimizer neither pushes the aggregation before the joins in the subquery nor moves it after any joins outside the scope of the subquery. If the GROUP BY clause is on the main query, the optimizer performs the aggregation after joining all tables. Once again views and inline TVFs behave identically to FROM clause subqueries with regards to aggregation placement and the FORCE ORDER hint.

Let's look at an example of how we can use the FORCE ORDER hint to control the placement of an aggregation operator. Consider the following query, which counts the number of orders placed by customers from the United States:

```
SELECT O.[CustomerId], COUNT(*)
FROM [Customers] C JOIN [Orders] O
    ON C.[CustomerId] = O.[CustomerId]
WHERE C.[Country] = N'USA'
GROUP BY O.[CustomerId]
```

The optimizer pushes the aggregation below the join, as you can see in [Figure 4-18](#).

Figure 4-18. Query plan for a GROUP BY operation that aggregates before joining



This plan first computes the number of orders for all customers and then joins the results of the aggregation with the Customers table to filter out those customers that don't live in the United States. Suppose that we would rather perform the join and filtering before we compute the aggregation. We can do that with the FORCE ORDER hint:

```
SELECT O.[CustomerId], COUNT(*)
FROM [Customers] C JOIN [Orders] O
    ON C.[CustomerId] = O.[CustomerId]
WHERE C.[Country] = N'USA'
GROUP BY O.[CustomerId]
```

[Figure 4-19](#) shows the plan with the aggregation pulled to the top.

Figure 4-19. Query plan for a GROUP BY operation with a FORCE ORDER hint



Note that there is no way to force the aggregation placement without forcing the join order as well. We were careful in this example to force the optimizer to join the Customers table before the Orders table by listing the Customers table first. This join order results in an efficient nested loops join. Had we forced the opposite join order by listing the tables in the opposite order, the optimizer could not have used the nested loops join and would have been forced to use a far less efficient many-to-many merge join.

MAXDOP N Hint

The MAXDOP N hint can be used to change the degree of parallelism used both to optimize and to execute the query. If the max degree of parallelism server level configuration option and the MAXDOP hint are both used, the MAXDOP hint overrides the server level configuration option. MAXDOP 0â—the defaultâ—specifies that SQL Server should decide the degree of parallelism, MAXDOP 1 forces a serial plan (no parallelism), and any other value allows a degree of parallelism of up to N. The query processor estimates the degree of parallelism at compile time based on the number of available processors, the affinity mask, max degree of parallelism, and max worker threads global configuration options, and any value specified in the MAXDOP hint. It takes this expected degree of parallelism into account during optimization. The best serial plan may be different from the best parallel plan and the best parallel plan for a low degree of parallelism may be different from the best parallel plan at a higher degree of parallelism. For example, operators that depend on ordered data, such as merge join or stream aggregate, tend not to scale well and tend to perform poorly at higher degrees of parallelism. Thus, as the degree of parallelism increases, the optimizer is less likely to choose a plan with a parallel merge join or parallel stream aggregate.

The expected degree of parallelism used during optimization may differ from the actual degree of parallelism used during execution. As discussed in [Chapter 3](#), SQL Server chooses the actual degree of parallelism immediately before beginning execution. The MAXDOP N hint does not guarantee the actual degree of parallelism will be N and does not guarantee that the degree of parallelism used during optimization will match the degree of parallelism used at execution.

EXPAND VIEWS Hint

This hint is only useful on the Enterprise Edition of SQL Server. On Enterprise Edition, if we submit a query that exactly or partially matches an indexed view, the optimizer may substitute the indexed view for the portion of the original query that matches the view. The EXPAND VIEWS hint prevents the optimizer from matching indexed views.

Note that when we submit a query that references an indexed view, SQL Server (all editions) expands the view using its definition. This expansion occurs early during query processing. On Enterprise Edition, the optimizer may subsequently match the expanded view definition and replace it with the indexed view. While in some cases it may appear as though nothing has happened (that is, the view appears in the query plan just as it appeared in the query text), the view was actually expanded and subsequently matched. The EXPAND VIEWS hint prevents view matching regardless of whether the view or its definition appeared in the original query. Thus, this hint also has the effect of ensuring the views are expanded but not subsequently matched.

Table Hints

Three table hints affect plan selection: INDEX, NOEXPAND, and FASTFIRSTROW. Unlike query hints, table hints apply to a single table and can be used within subqueries, views, and inline TVFs. The examples using table hints will show text-based execution plans instead of graphical plans because it is much easier to see the details of which indexes were used in the text version.

INDEX Hint

An INDEX hint allows us to force the query optimizer to use a specific index (or the heap) for a scan or seek. This hint can be useful to force or to avoid a bookmark lookup or even to force an index intersection. To force the optimizer to use a specific index, we can identify the desired index by index_id (which you can find in the sys.indexes catalog view) or by name. With the exception of index_id values 0 and 1 which always refer to the heap and clustered indexes respectively, it is generally much simpler and safer to refer to indexes by name. We have control over the names of indexes but do not have control over the index_id values that SQL Server assigns to each index.

Note



Although you can use an INDEX hint to force the optimizer to use any index, you cannot force the optimizer to choose an index seek instead of an index scan. However, in practice, if you have a seekable predicate on the index keys of the specified index, the optimizer will usually choose the index seek.

Moreover, while you can always force a clustered index or table scan using an INDEX(0) hint, you cannot force a nonclustered index scan. Of course, if you do not have a seekable predicate on the index keys of the specified index, you will get an index scan as the optimizer has no other alternative. However, there are few, if any, reasons why one would want to force an index scan when an index seek (over the same index) is possible.

Eliminating a Bookmark Lookup

Let's look at some examples of where and how we might use the INDEX hint. Suppose we have a plan with a nonclustered index seek and bookmark lookup, where the optimizer has underestimated the cardinality of the seek operator. Since the bookmark lookup consists of many random I/Os, this plan may be overly expensive. We might use an INDEX hint to force the optimizer to choose a covering index (or the heap), even if it means performing a full table scan. For example, consider the following query:

```
SELECT [OrderId], [CustomerId]
FROM [Orders]
WHERE [ShipPostalCode] = N'99362'
```

The postal code in this query occurs in only two rows. Thus, the query optimizer rightly (as there is no cardinality estimation error in this example) generates a plan that uses a bookmark lookup. The text plan makes it much easier to see what index is actually being used.

```
--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
--Index Seek(OBJECT:([Orders].[ShipPostalCode]),
SEEK:([Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)
```

```
--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)
```

Using an index hint, we can force the optimizer to choose a clustered index scan instead of a bookmark lookup (although in this case we are probably better off with the original plan):

```
SELECT [OrderId], [CustomerId]
FROM [Orders] WITH
WHERE [ShipPostalCode] = N'99362'
```

Since we do not have a predicate on the clustering key (OrderId), the optimizer honors the hint by generating a clustered index scan:

```
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]),
WHERE:([Orders].[ShipPostalCode]=N'99362'))
```

Forcing a Bookmark Lookup

Now suppose we have a plan with a full clustered index or table scan. We can use an INDEX hint to force the optimizer to choose a nonclustered index seek and bookmark lookup. We might prefer the index seek and bookmark lookup plan for several reasons. It might be cheaper if the optimizer has overestimated the cardinality of the seek operator. It might also be cheaper if we know that the base table is always in memory so that the bookmark lookups do not actually generate any physical I/Os. We also might prefer this plan because we wish to reduce the locking overhead or deadlock risk of the query by touching fewer rows.

Consider the same query from the prior example, but let's use a variable for the lookup value.

```
DECLARE @ShipCode nvarchar(20)
SET @ShipCode = N'99362'
SELECT [OrderId], [CustomerId]
FROM [Orders]
WHERE [ShipPostalCode] = @ShipCode
```

Now the optimizer, lacking a specific value for @ShipCode during optimization, must guess at the cardinality. It guesses a bit on the high side and chooses the clustered index scan, this time, without any hints:

```
--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]),
WHERE:([Orders].[ShipPostalCode]=[@ShipCode]))
```

If we know that all or virtually all executions of this statement will use a value for @ShipCode that retrieves relatively few rows, we might want to force the plan that uses a nonclustered index seek with a bookmark lookup:

```
DECLARE @ShipCode nvarchar(20)
SET @ShipCode = N'99362'
SELECT [OrderId], [CustomerId]
FROM [Orders]
```

```
WHERE [ShipPostalCode] = @ShipCode
```

With the INDEX hint, we get our desired plan:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
 |--Index Seek(OBJECT:([Orders].[ShipPostalCode]),
    SEEK:([Orders].[ShipPostalCode]=[@ShipCode]) ORDERED FORWARD)
 |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
    SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)
```

Multiple Nonclustered Indexes

The INDEX hint can also be useful to select a particular nonclustered index if a query has multiple predicates, each of which can be satisfied with an index seek over a different nonclustered index. As with the above examples, you might choose to override the optimizer's choice of index because of a cardinality estimation error or to reduce the risk of lock contention or a deadlock. The INDEX hint can even be used to force an index intersection by specifying more than one index in the hint. For instance, consider the following query:

```
SELECT [OrderId], [CustomerId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26' AND [ShipPostalCode] = N'99362'
```

The optimizer chooses a nonclustered index seek on the ShipPostalCode index with a bookmark lookup. It applies the OrderDate predicate as a residual predicate using a filter operator. The plan looks as follows:

Code View:

```
|--Filter(WHERE:([Orders].[OrderDate]='1998-02-26'))
 |--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
    |--Index Seek(OBJECT:([Orders].[ShipPostalCode]),
       SEEK:([Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
       SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)
```

While this plan is quite reasonable, we could choose to force an index intersection by including two indexes in the hint:

```
SELECT [OrderId], [CustomerId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26' AND [ShipPostalCode] = N'99362'
```

The resulting plan looks as follows:

Code View:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
 |--Merge Join(Inner Join, MERGE:([Orders].[OrderID])=([Orders].[OrderID]),
    RESIDUAL:(...))
```

```

|     |--Index Seek(OBJECT:([Orders].[OrderDate]),
|         SEEK:([Orders].[OrderDate]='1998-02-26') ORDERED FORWARD)
|     |--Index Seek(OBJECT:([Orders].[ShipPostalCode]),
|         SEEK:([Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)
|--Clustered Index Seek(OBJECT:([Orders].[PK_Orders])),
    SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)

```

Index Union

We cannot use an INDEX hint to force the optimizer to generate an index union plan. However, if we have at least one unique index (clustered or nonclustered) on the table, we can rewrite a query to force a union plan that closely mimics an index union. For example, consider the following query (which by default uses a full clustered index scan):

```

SELECT [OrderId], [CustomerId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26' OR [ShipPostalCode] = N'83720'

```

We can rewrite this query as a union:

```

SELECT [OrderId], [CustomerId]
FROM [Orders] WITH (INDEX([OrderDate]))
WHERE [OrderDate] = '1998-02-26'
UNION
SELECT [OrderId], [CustomerId]
FROM [Orders] WITH (INDEX([ShipPostalCode]))
WHERE [ShipPostalCode] = N'83720'

```

Note that once we rewrite the query, we may or may not need to include the index hints. We also have the choice of using one of the UNION hints, which we discussed earlier in this chapter to force a merge union, hash union, or concatenation with a distinct operator. The rewritten query yields the following plan:

Code View:

```

|--Merge Join(Union)
|--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
|     |--Index Seek(OBJECT:([Orders].[OrderDate]),
|         SEEK:([Orders].[OrderDate]='1998-02-26') ORDERED FORWARD)
|     |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
|         SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)
|--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID], [Expr1008])
    WITH ORDERED PREFETCH)
    |--Index Seek(OBJECT:([Orders].[ShipPostalCode]),
        SEEK:([Orders].[ShipPostalCode]=N'83720') ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
        SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)

```

This plan is not exactly identical to an index union because of the placement of the bookmark lookups. In this

plan, there are two bookmark lookups, one for each nonclustered index seek, and the bookmark lookups are performed before the union. In a true index union plan, there would be a single bookmark lookup after the index union operation.

Note that we must include the unique column(s) (in this case OrderId) in the select list to ensure that duplicate rows are correctly eliminated. Without the unique column(s), we might incorrectly end up eliminating duplicate values from different rows. For instance, if in the above query we simply wrote `SELECT [CustomerId]`, we would incorrectly eliminate duplicate CustomerIds from the result set. Technically, we do not need a unique index as long as we know that the set of columns in the select list are unique, however, the unique index guarantees that the rewritten query is semantically equivalent to the original query.

Detecting INDEX Hints in a Query Plan

We can detect that an INDEX hint was specified by looking at the Properties window along a graphical query plan, or by using `SHOWPLAN_XML` and looking for the `ForcedIndex` attribute. For example:

```
<RelOp NodeId="0" PhysicalOp="Clustered Index Scan"
        LogicalOp="Clustered Index Scan" ...>
  <IndexScan Ordered="0" NoExpandHint="0">
</RelOp>
```

NOEXPAND Hint

As we noted previously during the discussion of the EXPAND VIEWS query hint, when we submit a query that references an indexed view, SQL Server expands the view using its definition. The NOEXPAND table hint prevents SQL Server from expanding an explicitly referenced indexed view. This hint guarantees that SQL Server generates a query plan using the indexed view. Since views are never expanded, this hint overrides the EXPAND VIEWS query hint, which prevents view matching. Moreover, since only Enterprise Edition supports indexed view matching, this hint is also the only way to write a query that uses an indexed view for other editions of SQL Server.

FASTFIRSTROW Hint

Specifying the FASTFIRSTROW hint on any table in a query is exactly equivalent to specifying a `FAST 1` query hint for the entire query. There is little if any reason to use the FASTFIRSTROW hint. The `FAST N` query hint is more flexible as you can vary the value of `N`, and it is more intuitive as it makes it clear that the hint impacts not just one table but the entire query. If you use both a `FASTFIRSTROW` hint and a `FAST N` hint in the query, the `FAST N` hint takes precedence. That is, the optimizer uses the value of `N` from the explicit `FAST N` hint. Refer to the discussion of the `FAST N` hint earlier in this chapter for more information.

ANSI-Style Join Hints

ANSI-style join hints allow for fine-grained control over both the join order and the join type (in other words, the physical join operator). These hints are very powerful. We can force nearly any legal join order including bushy plans, we can force aggregation placement, and, more importantly, we can individually select the join type for each and every join. Although join hints do not allow us to control all operators, we can use join hints to handcraft a significant portion of a query plan.

Join hints are only available with ANSI-style joins as the join hint is specified between the logical join type keyword (`INNER`, `LEFT OUTER`, `RIGHT OUTER`, etc.) and the `JOIN` keyword. Like table hints, join hints

can be used within subqueries, views, and inline TVFs. However, if an ANSI-style join hint (including one hidden in a view or inline TVF) conflicts with a query join hint, optimization will fail.

Note that, as with the FORCE ORDER hint, with outer joins some join orders may not be possible without changing the semantics of the original query. Also, it may not be possible to force all join types. For example, we cannot force a merge or hash join for a join that lacks an equijoin predicate. If you try to force an illegal join type, optimization fails.

Tip



Using even one join hint anywhere in a query has the same effect as using the FORCE ORDER query hint. This means that if a query includes a join hint and also includes "unhinted" joins, the order of all of the joins including both the hinted and unhinted joins is forced to the order that the tables appear in the FROM clause as described in the earlier discussion of the FORCE ORDER hint. It also means that if a query has any subqueries or views with aggregations, the optimizer will not move these aggregations earlier or later in the plan. Thus, you should use join hints just as cautiously as the FORCE ORDER hint and watch out for the same pitfalls such as introducing a Cartesian product within a view.

Let's look at an example of how we can use join hints. Let's begin with the example from the FORCE ORDER hint discussion earlier in this chapter. Recall that we were able to force any join order including a right deep plan. However, with query-level join hints, we could not individually control the join types of each join. Now, suppose that we start with the right deep plan from this example, but we want to force the topmost join (between the Employees table and the result of the join of the Customers and Orders tables) to use a merge join rather than a hash join. Suppose that we also want to ensure that the join between the Customers and Orders tables continues to use a nested loops join. We can achieve this result with join hints as follows:

```
SELECT O.[OrderId]
FROM [Employees] E INNER JOIN
(
    [Customers] C INNER JOIN
    [Orders] O
    ON O.[CustomerId] = C.[CustomerId]
)
ON O.[EmployeeId] = E.[EmployeeId]
WHERE
    C.[City] = N'London' AND
    E.[LastName] = N'Peacock'
```

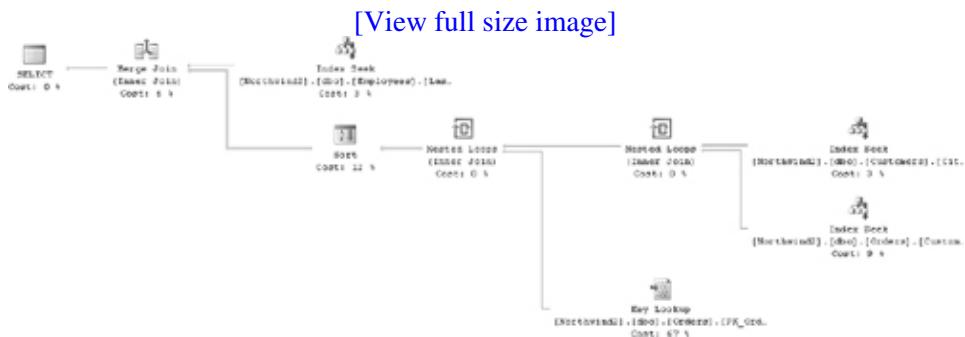
Notice that with the join hints we must specify the INNER keyword, even though it is the default for ANSI joins. Notice also that we do not need to specify the FORCE ORDER query hint as the join hints imply it. We do get the following warning message during compilation, which tells us that the optimizer is forcing the order of the joins because of the join hints:

Code View:

Warning: The join order has been enforced because a local join hint is used.

As expected, we get the following query plan, as shown in [Figure 4-20](#).

Figure 4-20. Query plan for a using two ANSI JOIN hints



As we discussed earlier, because merge joins require that data be sorted, by forcing a merge join, we have also forced the optimizer to introduce a sort.

USE PLAN Hint

The USE PLAN hint is new to SQL Server 2005. With the USE PLAN hint we can tell the optimizer exactly which plan we want by simply providing the XML plan to be use. This hint is capable of forcing nearly all aspects of a query plan, including the choice of scans versus seeks, index choices, join orders and types, aggregation placements and types, and the decision of whether to use a serial or parallel plan. The USE PLAN hint is ideally suited for "freezing" a previously generated plan and ensuring that SQL Server uses it each time a query is executed. Like the join hints described earlier, it can also be useful for capturing a plan on a production system and forcing it for testing or troubleshooting purposes on a development system. Although the USE PLAN hint can be used directly on any query, it is especially useful in conjunction with plan guides, which we discuss in [Chapter 5](#). While the USE PLAN hint is extremely powerful and can, in principle, be used to force nearly any legitimate query plan, it does have some drawbacks and limitations:

- The XML plan provided as "input" to the USE PLAN hint must be schema valid. SQL Server uses the published XML plan schema to validate all USE PLAN input. If it finds any invalid input, it generates an XML validation error and fails without generating any query plan at all.
- USE PLAN, like all other optimization hints, cannot be used to force the optimizer to generate an invalid query plan or a query plan that it would not otherwise consider during normal optimization. The latter limitation is necessary for SQL Server to ensure that the plan you provide with the USE PLAN hint is a valid plan that will return correct results for the query in question. If you attempt to force an invalid plan or a plan that the optimizer would not normally consider, SQL Server generates an error and the query does not execute.
- Schema changes (for example, dropping an index or constraint) may render a once-valid plan invalid and cause a query that tries to force the now-invalid plan via a USE PLAN hint to fail. It is a good practice after any schema change to confirm that all forced plans are still valid and that all USE PLAN hints continue to work as expected.
- XML plans are quite verbose and quite complex. Compared to the other hints described earlier in this chapter, a USE PLAN hint is difficult to read or interpret although it can be viewed graphically using Management Studio. It is extremely challenging to generate an XML plan by hand that is schema valid, semantically valid, and represents a query plan that is both valid for a particular query and can be generated by the optimizer. Because it is so hard to generate an XML plan manually, the USE PLAN hint is not well suited or recommended for experimenting with alternative plans. A better alternative and best practice is to use the hints described earlier in this chapter to experiment with or explore various plan alternatives. It is much easier to work with these hints than to work with XML plans directly. Once you find a plan that you like, use the SET SHOWPLAN_XML option to have SQL Server generate the XML plan for use with a USE PLAN hint.

- SQL Server will not cache query plans that include string literals that exceed 8 KB in length. If the XML plan string, which you provide as input to the USE PLAN hint, exceeds 8 KB in length, SQL Server cannot cache the query plan. Given how verbose XML plans are, the USE PLAN hint will push all but the simplest queries over the threshold. If a query plan cannot be cached, it must be recompiled on each execution. These extra recompilations are inefficient and will drive up the cost of processing the query. Plan guides (described in [Chapter 5](#)) provide a workaround to this problem.
- While the USE PLAN hint can be used to freeze plans even across server upgrades, there are no absolute guarantees that this use of the USE PLAN hint will work. A future version of the server may no longer be able to generate the same plan used by an older version. Recall again that no optimization hint, not even USE PLAN, can force the optimizer to generate a plan that it would not otherwise consider. Each new version of SQL Server introduces many new features, including functional and performance improvements. Many of these improvements result in significant changes to the space of plans that the optimizer explores and generates.
- While most of the hints discussed can be used to force plans both for SELECT statements (including cursors) as well as for INSERT, UPDATE, and DELETE statements, the USE PLAN hint can only be used to force plans for SELECT statements (including cursors). It cannot be used to force plans for INSERT, UPDATE, or DELETE statements.
- Ordinary SELECT statements and cursors defined for the same SELECT generate different query plans. Thus, even if a SELECT statement and a cursor execute the same query and return the same data, the XML plan from the SELECT statement can only be used in a USE PLAN hint with the SELECT statement and the XML plan from the cursor can only be used in a USE PLAN hint with the cursor. Moreover, a USE PLAN hint can only be used with static and fast-forward-only cursors. A USE PLAN hint cannot be used with dynamic or keyset cursors.
- The USE PLAN hint cannot be combined with any of the other plan forcing query hints described earlier in this chapter. Any attempt to combine the USE PLAN hint with another plan forcing query hint (for example, FORCE ORDER, LOOP JOIN, ORDER GROUP, etc.) results in an error. This restriction is fairly minor because the USE PLAN hint enables us to force nearly any plan without the use of other hints.
- It is possible to combine the USE PLAN hint with table and ANSI syntax join hints. For example, you can use the USE PLAN hint on a query that includes a view which includes an INDEX or ANSI join hint without changing the view definition. In this case, the query succeeds and the USE PLAN hint overrides the other hints.
- While the USE PLAN hint can be used to force most aspects of a plan, there are a few operators that cannot be forced even with the USE PLAN hint. These operators include Assert, Bitmap, and Compute Scalar. The USE PLAN hint also cannot change the memory grant allotted to a query (which is determined by the actual cardinality estimates and available memory) or the degree of parallelism for a parallel query (which is determined by MAXDOP settings and by available threads).

XML plans are too verbose to include an entire USE PLAN hint in an example. However, you can try the following simple exercise to see how USE PLAN works. Begin by collecting the XML plan for the same query that we used for the ANSI-style join hint example:

```
SET SHOWPLAN_XML ON
GO

SELECT O.[OrderId]
FROM [Employees] E INNER JOIN
(
    [Customers] C INNER JOIN
    [Orders] O
    ON O.[CustomerId] = C.[CustomerId]
)
ON O.[EmployeeId] = E.[EmployeeId]
WHERE
    C.[City] = N'London' AND
    E.[LastName] = N'Peacock'
GO
```

```
SET SHOWPLAN_XML OFF
GO
```

Note that the SET SHOWPLAN_XML statements must be executed in their own batches. Next, cut and paste the entire XML plan output into a USE PLAN hint on the same query without the join hints. This is easiest to do if you set the query options in SQL Server Management Studio to send the results to a grid, right-click the grid cell containing the XML plan, and select copy. If you send the results to text, SQL Server Management Studio may truncate the XML output. Run the query with the USE PLAN hint and check the query plan again (using whatever means you like—text, graphical, or XML). Notice that this query yields the same plan as the query with the join hints.

```
SELECT O.[OrderId]
FROM [Employees] E JOIN
(
    [Customers] C JOIN
    [Orders] O
    ON O.[CustomerId] = C.[CustomerId]
)
ON O.[EmployeeId] = E.[EmployeeId]
WHERE
    C.[City] = N'London' AND
    E.[LastName] = N'Peacock'
```

Finally, we can detect that the query plan was forced by a USE PLAN hint by checking for the UsePlan attribute under the <QueryPlan> element in the XML plan.

Query Processing Best Practices

In this final section, we'll present some general recommendations for getting the best performance from your queries. You've probably seen most of these recommendations before, but they are so important, that it can't hurt to be reminded.

Use Set-Oriented Programming

This first suggestion has been mentioned earlier in this chapter, but it is probably the number-one query performance issue with new SQL Server developers. Any code written to examine and manipulate a set of rows should be implemented using Transact-SQL's own set-based operations. Defining a cursor to step through the rows in a table is a giant step backwards, and causes you to lose most of the power of the SQL Server data processing engine.

Defining a cursor to step through rows is not good database programming practice for many reasons, including the following:

- It is less efficient than the SQL Server's built-in nested loops iterator, because it requires a much deeper execution stack for each row.
- It does not allow the query optimizer to determine an alternate join order, or different execution algorithms, execute the query in a different order, or with different execution algorithms.
- It creates more complex code that is more difficult to understand, optimize, and maintain.

Although defining your processing using cursors can in some cases make it easier for a new database developer to get the desired results, it does not allow for the higher-level, declarative power of the T-SQL

language.

Provide the Optimizer with Constraints and Statistics Information

The plans produced by the query optimizer depend heavily on the quality of information it has available to achieve good performance and robustness. There are two main sources of information you can provide to help the optimizer make the best choices:

- **Constraints** SQL Server constraints that can affect the query plan include those for uniqueness, foreign keys, and check conditions. You should always declare the constraints that your data needs to satisfy. If the optimizer then knows that the data meets certain condition, it can generate more specific plans. Declarative constraints provide much more benefit to the optimizer than any validation you do through stored procedure, function or trigger code.
- **Statistics** Run SQL Server with AUTO_UPDATE_STATISTICS turned on in all databases, when it is possible. If you must, switch to manual maintenance; be sure to re-enable AUTO_UPDATE_STATISTICS when you can. If you have no statistics, or unreliable statistics, that optimizer is likely to make poor choices for your query plans.

Avoid Unnecessary Complexity

We have already seen several types of query constructs for which the optimizer will not be able to always come up with the best plan. We'll review some of them here:

- Very large queries or constructs that do not limit the possible results. When there are insufficient filters so that the set of possible results gets sufficiently large, the optimizer relies more on heuristics to "guess" what a good might would be. Therefore, some execution plan alternatives that might be more efficient might not be considered at all by the optimizer. Typically, the optimizer can process JOIN operations much better than multiple GROUP BY clauses and subqueries.
- Constructs for which the optimizer cannot easily estimate the number of qualifying rows or the distribution of the data. These constructs include aggregations, scalar expressions, UDFs, and multistatement TVFs.

These types of constructs can reduce the reliability of the optimizer's plan selection algorithms, so be careful when you use them. The following best practices should be considered to help the optimizer generate more reliable and robust plans:

- Create computed columns for scalar expression that your queries will use to filter the results.
- Use complex constructs that will be applied as one of the last steps of your query's plan, wherever possible. Errors in estimation of cardinality or data distribution early in the plan will affect the choice of operators that come later.
- Split your complex queries into multiple steps, and store intermediate results in temporary tables.

Be Careful with Dynamic SQL

SQL Server permits a number of dynamic language features that you may occasionally find very useful. The following is a list of some of these features:

- **Unqualified object names** Objects in different schemas can have the same name. If a query uses an unqualified schema name, the object is resolved based on the default schema of the user that executes the query. This behavior can introduce additional processing requirements during plan generation, as the actual object may be unknown. If it is not critical to your application to have dynamic name

resolution, you will usually be better off if you qualify all object names with the schema containing them. For example, you should refer to a table as dbo.Customers instead of simply Customers.

- Temporary tables with unqualified schema A temporary table is always dynamically resolved at execution time, so that it is possible for the same batch or procedure to create temporary tables in different schemas. This capability is a more extreme case of the dynamic name resolution situation mentioned in the previous paragraph. A best practice is to make sure that the schema for temporary tables is not left ambiguous.
- Dynamically generated SQL statements The ability to generate dynamic SQL code can be considered to be one of the strengths of relational databases. However, there are security risks in creating your SQL statements dynamically. Dynamic SQL that is issued from within a stored procedure will typically execute using the security context of the caller, not the owner of the stored procedure. To allow all callers to execute the dynamic code, you might then need to grant those users a higher level of security than you would if dynamic SQL was not used. SQL injection is also always a risk when the dynamic SQL code is generated based on user input. A best practice is to avoid the use of dynamic SQL code if at all possible.
- Changing connection settings Connection SET options such as ANSI_NULLS or ARITH_ABORT that affect the semantics of your queries can also affect the plans that are generated for those queries. Re-executing the same set of queries after changing SET options negatively impacts performance by requiring recompilation and reducing plan reuse. A best practice is to keep the default SQL Server connection settings across all applications.

Summary

In this chapter, we examined the role of optimization in query processing and discussed some of the steps that the optimizer goes through in coming up with an execution plan for your queries. Keep in mind that for all but the simplest queries, there are many possible plans that the optimizer could evaluate. In order to be able to optimize in a reasonable amount of time, SQL Server's optimizer may stop searching for better plans as soon as it finds one that it considers "good enough." The vast majority of the time, the SQL Server query processor works extremely well. The optimizer generally chooses satisfactory query plans, and most applications just work. For those occasions where problems do arise, we looked at a variety of issues and pitfalls that can affect query performance. We've discussed how to spot and handle cardinality estimation errors as well as query improvements and potential schema changes that can boost query performance. Finally, we discussed the hints that SQL Server offers to help users diagnose, troubleshoot, and, as a last resort, work around various query performance issues.

There is no one-size-fits-all strategy for diagnosing and fixing query performance issues. The hope is that, with a basic understanding of how query processing works along with the tools for viewing your query plans and hints for controlling your query plans, you will be able to troubleshoot and fix or work around a wide variety of new and/or unexpected issues.

Chapter 5. Plan Caching and Recompilation

â Kalen Delaney

In this chapter:	
The Plan Cache	277
Caching Mechanisms	279
Plan Cache Internals	300

Objects in Plan Cache: The Big Picture	311
Multiple Plans in Cache	313
When to Use Stored Procedures and Other Caching Mechanisms	314
Troubleshooting Plan Cache Issues	315
Summary	330

We've now looked at SQL Server's query optimization process and the details of query execution. Because query optimization can be a complex and time-consuming process, SQL Server frequently and beneficially reuses plans that have already been generated and saved in the plan cache, rather than producing a completely new plan. However, in some cases a previously created plan may not be ideal for the current query execution, and we might achieve better performance by creating a new plan.

In this chapter, we'll look at the SQL Server 2005 plan cache and how it is organized. We'll tell you about what kinds of plans are saved, and under what conditions SQL Server might decide to reuse them. We'll look at what might cause an existing plan to be re-created. We'll also look at the metadata that describes the contents of plan cache. Finally, we'll describe the ways that you can encourage SQL Server to use an existing plan when it might otherwise create a new one, and how you can force SQL Server to create a new plan when you need to know that the most up-to-date plan is available.

The Plan Cache

It's important to understand that the plan cache in SQL Server 2005 is not actually a separate area of memory. Releases prior to SQL Server 7 had two effective configuration values to control the size of the plan cache, which was then called the procedure cache. One specified a fixed size for SQL Server's total usable memory; the other specified a percentage of that memory (after fixed needs were satisfied) to be used exclusively for storing procedure plans. Also, in releases prior to SQL Server 7, query plans for adhoc SQL statements were never stored in the cache, only the plans for stored procedures. That is why it was called procedure cache in older versions. In SQL Server 2005, the total size of memory is by default dynamic, and the space used for query plans is also very fluid.

Plan Cache Metadata

In the first part of this chapter, we'll explore the different mechanisms by which a plan can be reused, and to observe this plan reuse (or nonreuse) we only need to look at a couple of different metadata objects. There are actually about a dozen different metadata views and functions that give us information about the contents of plan cache, and that doesn't include the metadata that gives us information about memory usage by plan cache. Later in the chapter, we'll look at more details available in the plan cache metadata, but for now we'll be using just one view and one function. The view is sys.dm_exec_cached_plans, which contains one row for each plan in cache, and we'll look at the columns usecounts, cacheobjtype, and objtype. The value in usecounts will allow us to see how many times a plan has been reused. The possible values for cacheobjtype and objtype will be described in the next section. We'll also use the value in the column plan_handle as the parameter when we use the new SQL Server 2005 CROSS APPLY capability to join the sys.dm_exec_cached_plans view with the table-valued function (TVF) sys.dm_exec_sql_text. This is the query we'll use, which we'll refer to as the usecount query:

```
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
```

```
AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

Clearing Plan Cache

Because SQL Server 2005 has the potential to cache almost every query, the number of plans in cache can become quite large. There is a very efficient mechanism, described later in the chapter, for finding a plan in cache. There is not a direct performance penalty for having lots of cached plans, aside from the memory usage. However, if you have many very similar queries, the lookup time for SQL Server to find the right plan can sometimes be excessive. In addition, from a testing and troubleshooting standpoint, having lots of plans to look at can sometimes make it difficult to find just the plan in which we're currently interested. SQL Server provides a mechanism for clearing out all the plans in cache, and you will probably want to do that occasionally on your test servers to keep the cache size manageable and easy to examine. You can use either of the following commands:

- DBCC FREEPROCCACHE

This command removes all cached plans from memory

- DBCC FLUSHPROCINDB (<dbid>)

This command allows you to specify a particular database id, and then clears all plans from that particular database. Note that the usecount query that we'll use in this section does not return database id information, but the sys.dm_exec_sql_text TVF has that information available, so dbid could be added to the usecount query.

It is, of course, recommended that you don't use these commands on your production servers, as it could impact the performance of your running applications. Usually, you want to keep plans in cache.

Chapter 5. Plan Caching and Recompilation

â Kalen Delaney

In this chapter:	
The Plan Cache	277
Caching Mechanisms	279
Plan Cache Internals	300
Objects in Plan Cache: The Big Picture	311
Multiple Plans in Cache	313
When to Use Stored Procedures and Other Caching Mechanisms	314
Troubleshooting Plan Cache Issues	315
Summary	330

We've now looked at SQL Server's query optimization process and the details of query execution. Because query optimization can be a complex and time-consuming process, SQL Server frequently and beneficially

reuses plans that have already been generated and saved in the plan cache, rather than producing a completely new plan. However, in some cases a previously created plan may not be ideal for the current query execution, and we might achieve better performance by creating a new plan.

In this chapter, we'll look at the SQL Server 2005 plan cache and how it is organized. We'll tell you about what kinds of plans are saved, and under what conditions SQL Server might decide to reuse them. We'll look at what might cause an existing plan to be re-created. We'll also look at the metadata that describes the contents of plan cache. Finally, we'll describe the ways that you can encourage SQL Server to use an existing plan when it might otherwise create a new one, and how you can force SQL Server to create a new plan when you need to know that the most up-to-date plan is available.

The Plan Cache

It's important to understand that the plan cache in SQL Server 2005 is not actually a separate area of memory. Releases prior to SQL Server 7 had two effective configuration values to control the size of the plan cache, which was then called the procedure cache. One specified a fixed size for SQL Server's total usable memory; the other specified a percentage of that memory (after fixed needs were satisfied) to be used exclusively for storing procedure plans. Also, in releases prior to SQL Server 7, query plans for adhoc SQL statements were never stored in the cache, only the plans for stored procedures. That is why it was called procedure cache in older versions. In SQL Server 2005, the total size of memory is by default dynamic, and the space used for query plans is also very fluid.

Plan Cache Metadata

In the first part of this chapter, we'll explore the different mechanisms by which a plan can be reused, and to observe this plan reuse (or nonreuse) we only need to look at a couple of different metadata objects. There are actually about a dozen different metadata views and functions that give us information about the contents of plan cache, and that doesn't include the metadata that gives us information about memory usage by plan cache. Later in the chapter, we'll look at more details available in the plan cache metadata, but for now we'll be using just one view and one function. The view is sys.dm_exec_cached_plans, which contains one row for each plan in cache, and we'll look at the columns usecounts, cacheobjtype, and objtype. The value in usecounts will allow us to see how many times a plan has been reused. The possible values for cacheobjtype and objtype will be described in the next section. We'll also use the value in the column plan_handle as the parameter when we use the new SQL Server 2005 CROSS APPLY capability to join the sys.dm_exec_cached_plans view with the table-valued function (TVF) sys.dm_exec_sql_text. This is the query we'll use, which we'll refer to as the usecount query:

```
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

Clearing Plan Cache

Because SQL Server 2005 has the potential to cache almost every query, the number of plans in cache can become quite large. There is a very efficient mechanism, described later in the chapter, for finding a plan in cache. There is not a direct performance penalty for having lots of cached plans, aside from the memory usage. However, if you have many very similar queries, the lookup time for SQL Server to find the right plan can sometimes be excessive. In addition, from a testing and troubleshooting standpoint, having lots of plans to

look at can sometimes make it difficult to find just the plan in which we're currently interested. SQL Server provides a mechanism for clearing out all the plans in cache, and you will probably want to do that occasionally on your test servers to keep the cache size manageable and easy to examine. You can use either of the following commands:

- DBCC FREEPROCCACHE

This command removes all cached plans from memory

- DBCC FLUSHPROCINDB (<dbid>)

This command allows you to specify a particular database id, and then clears all plans from that particular database. Note that the usecount query that we'll use in this section does not return database id information, but the sys.dm_exec_sql_text TVF has that information available, so dbid could be added to the usecount query.

It is, of course, recommended that you don't use these commands on your production servers, as it could impact the performance of your running applications. Usually, you want to keep plans in cache.

Caching Mechanisms

SQL Server can avoid compilations of previously executed queries by using four mechanisms to make plan caching accessible in a wide set of situations.

- Adhoc query caching
- Autoparameterization
- Prepared queries, using either sp_executesql or the prepare and execute method invoked through your API
- Stored procedures or other compiled objects (triggers, TVFs, etc.)

To determine which mechanism is being used for each plan in cache, we need to look at the values in the cacheobjtype and objtype columns in the sys.dm_exec_cached_plans view. The cacheobjtype column can have one of five possible values:

- Compiled Plan
- Parse Tree
- Extended Proc
- CLR Compiled Func
- CLR Compiled Proc

In this section, the only value we'll be looking at is Compiled Plan. Notice that we filter usecount query to limit the results to rows with this value.

There are 11 different possible values for the objtype column:

- Proc (Stored procedure)
- Prepared (Prepared statement)
- Adhoc (Adhoc query)
- ReplProc (Replication-filter-procedure)
- Trigger ()
- View
- Default (Default constraint or default object)
- UsrTab (User table)
- SysTab (System table)

- Check (CHECK constraint)
- Rule (Rule object)

We'll be mainly examining the first three values, but many caching details that apply to stored procedures also apply to replication filter procedures and triggers.

Adhoc Query Caching

If the caching metadata indicates a cacheobjtype value of Compiled Plan and an objtype value of Adhoc, the plan is considered to be an adhoc plan. Prior to SQL Server 2005, adhoc plans were occasionally cached, but it was not something on which you could depend. However, even when SQL Server caches your adhoc queries, you might not be able to depend on their reuse. When SQL Server caches the plan from an adhoc query, the cached plan will be used only if a subsequent batch matches exactly. This feature requires no extra work to use, but it is limited to exact textual matches. For example, if the following three queries are executed in the Northwind2 database (which can be found on the companion website), the first and third queries will use the same plan, but the second one will need to generate a new plan:

```
SELECT * FROM Orders WHERE CustomerID = 'HANAR'
SELECT * FROM Orders WHERE CustomerID = 'CHOPS'
SELECT * FROM Orders WHERE CustomerID = 'HANAR'
```

You can verify this by first clearing out plan cache, and then running the three queries above, in separate batches. Then run the usecount query referred to above:

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'CHOPS';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

You should get two rows back, because the NOT LIKE condition filters out the row for the usecount query itself. The two rows are shown here and indicate that one plan was used only once, and the other was used twice:

Usecounts	Cacheobjtype	Objtype	Text
1	Compiled Plan	Adhoc	SELECT * FROM Orders WHERE CustomerID = 'CHOPS'
2	Compiled Plan	Adhoc	SELECT * FROM Orders WHERE CustomerID = 'HANAR'

The results show that with a change of the actual data value, the same plan cannot be reused. However, to take advantage of reuse of adhoc query plans, you need to make sure that not only are the same data values used in the queries, but that the queries are identical, character for character. If one query has a new line or an extra space that another one doesn't have, they will not be treated as the same query. If one query contains a

comment that the other doesn't have, they will not be identical. In addition, if one query uses a different case for either identifiers or keywords, even in a database with a case-insensitive collation, the queries will not be the same. If you run the code below, you will see that none of the queries can reuse the same plan.

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM orders WHERE customerID = 'HANAR'
GO
-- Try it again
SELECT * FROM orders WHERE customerID = 'HANAR'
GO
SELECT * FROM orders
WHERE customerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR'
GO
select * from orders where customerid = 'HANAR'
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

Your results should show five rows in sys.dm_exec_cached_plans, each with a usecounts value of 1.

Note



Note that the SELECT statements are all in their own batch, separated by GO. If there were not GOs, there would just be one batch, and each batch has its own plan containing the execution plan for each individual query within the batch. For reuse of adhoc query plans, the entire batch must be identical.

Autoparameterization

For certain queries, SQL Server can decide to treat one or more of the constants as parameters. When this happens, subsequent queries that follow the same basic template can use the same plan. For example, these two queries run in the Northwind2 database can use the same plan:

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 6;
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 2;
```

Internally, SQL Server parameterizes these queries as follows:

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = @1
```

You can observe this behavior by running the following code, and observing the output of the usecount query:

```
USE Northwind2
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

You should get three rows returned, similar to the following:

Usecounts	Cacheobjtype	Objtype	Text
1	Compiled Plan	Adhoc	SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2;
1	Compiled Plan	Adhoc	SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
2	Compiled Plan	Prepared	(@1 tinyint)SELECT [FirstName], [LastName], [Title] FROM [Employees] WHERE [EmployeeID] = @1

You should notice that the two individual queries with their distinct constants do get cached as adhoc queries. However, these are only considered shell queries and are only cached to make it easier to find the autoparameterized version of the query if the exact same query with the same constant is reused at a later time. These shell queries do not contain the full execution plan but only a pointer to the full plan in the corresponding prepared plan. The third row returned from sys.dm_exec_cached_plans has an objtype of prepared. The query plan is associated with the prepared plan, and you can observe that the plan was used twice. In addition, the text now shows a parameter in place of a constant.

By default, SQL Server is very conservative about deciding when to autoparameterize. SQL Server will autoparameterize queries only if the query template is considered to be safe. A template is safe if the plan selected will not change even if the actual parameters change. This ensures that autoparameterization won't degrade a query's performance. The employees table used in the queries above has a unique index, so any query that has an equality comparison on employeeID is guaranteed to never find more than one row, so a plan using a seek on that unique index can be useful no matter what actual value is used.

However, consider a query that has either an inequality comparison or an equality comparison on a nonunique column. In those situations, some actual values may return many rows, and others return no rows, or only one. A nonclustered index seek might be a good choice when only a few rows are returned, but a terrible choice

when many rows are returned. So a query for which there is more than one possible best plan, depending on the value used in the query, is not considered safe, and it will not be autoparameterized. By default, the only way for SQL Server to reuse a plan for such a query is to use the adhoc plan caching described in the previous section.

In addition to requiring that there only be one possible plan for a query template, there are many query constructs that normally disallow autoparameterization. Such constructs include any statements with the following elements:

- JOIN
- BULK INSERT
- IN lists
- UNION
- INTO
- FOR BROWSE
- OPTION <query hints>
- DISTINCT
- TOP
- WAITFOR statements
- GROUP BY, HAVING, COMPUTE
- Full-text predicates
- Subqueries
- FROM clause of a SELECT statement has table valued method or full-text table or OPENROWSET or OPENXML or OPENQUERY or OPENDATASOURCE
- Comparison predicate of the form EXPR <> a non-null constant

Autoparameterization is also disallowed for data modification statements that use the following constructs:

- DELETE/UPDATE with FROM CLAUSE
- UPDATE with SET clause that has variables

Forced Parameterization

If your application uses many similar queries that you know will benefit from the same plan, but are not autoparameterized, either because SQL Server doesn't consider the plans safe or because they use one of the disallowed constructs, SQL Server 2005 provides an alternative. A new database option called FORCED PARAMETERIZATION can be enabled with the following command:

```
ALTER DATABASE <database_name> SET PARAMETERIZATION FORCED;
```

Once this option is enabled, SQL Server will always treat constants as parameters, with only a very few exceptions. These exceptions as listed in the SQL Server 2005 Books Online include:

- INSERT . . . EXECUTE statements.
- Statements inside the bodies of stored procedures, triggers, or user-defined functions. SQL Server already reuses query plans for these routines.
- Prepared statements that have already been parameterized on the client-side application.
- Statements that contain XQuery method calls, in which the method appears in a context in which its arguments would typically be parameterized, such as a WHERE clause. If the method appears in a context in which its arguments would not be parameterized, the rest of the statement is parameterized.
- Statements inside a Transact-SQL cursor. (SELECT statements inside API cursors are parameterized.)
- Deprecated query constructs.

- Any statement that is run in the context of ANSI_PADDING or ANSI_NULLS set to OFF.
- Statements that contain more than 2,097 literals.
- Statements that reference variables, such as WHERE T.col2 >= @p.
- Statements that contain the RECOMPILE or OPTIMIZE FOR query hints.
- Statements that contain a COMPUTE clause.
- Statements that contain a WHERE CURRENT OF clause.

You need to be careful when setting this option on for the entire database, because assuming that all constants should be treated as parameters during optimization, and then reusing existing plans, frequently gives very poor performance. An alternative that allows only selected queries to be autoparameterized is to use plan guides, which will be discussed at the end of this chapter. In addition, plan guides can also be used to override forced parameterization for selected queries, if the database has been set to FORCED PARAMETERIZATION.

Drawbacks of Autoparameterization

A feature of autoparameterization that you might have noticed in the output from the usecount query above is that SQL Server makes its own decision as to the datatype of the parameter, which might not be the datatype you think should be used. In the earlier example, looking at the employees table, SQL Server chose to assume a parameter of type tinyint. If we rerun the batch, and use a value that doesn't fit into the tinyint range (that is, a value less than 0 or larger than 255), SQL Server will not be able to use the same autoparameterized query. The batch below autoparameterizes both SELECT statements, but it is not able to use the same plan for both queries. The output from the usecount query should show two adhoc shell queries, and two prepared queries. One prepared query will have a parameter of type tinyint and the other will be smallint. As strange as it may seem, even if you switch the order of the queries, and use the bigger value first, you will get two prepared queries with two different parameter datatypes.

```
USE Northwind2
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 622;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

The only way to force SQL Server to use the same datatype for both queries is to enable FORCED PARAMETERIZATION for the database.

As mentioned, autoparameterization is not always appropriate, which is why SQL Server is so conservative in choosing to use it. Consider the following example. The BigOrders table in the Northwind2 database has 4,150 rows and 105 pages, so we might expect that a table scan reading 105 pages would be the worst possible performance for any query accessing the BigOrders table. There is a nonclustered nonunique index on the CustomerID column. If we enabled forced parameterization for the Northwind2 database, the plan used for the first SELECT will also be used for the second SELECT, even though the constants are different. The first query returns 5 rows and the second returns 155. Normally, a nonclustered index would be chosen for the first SELECT, and a clustered index scan for the second, because the number of qualifying rows exceeds the number of pages in the table. However, with PARAMETERIZATION FORCED, that's not what we get, as

you can see when you run the code below.

```
USE Northwind2
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION FORCED
GO
SET STATISTICS IO ON
GO
DBCC FREEPROCCACHE
GO
SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'
GO
SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION SIMPLE
GO
```

When we run the above code, we see that the first SELECT required 12 logical reads and the second required 312, almost three times as many reads as would have been required if scanning the table. The output of the usecount query shows that forced parameterization was applied and the parameterized prepared plan was used twice:

Usecounts	Cacheobjtype	Objtype	Text
1	Compiled Plan	Adhoc	SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'
1	Compiled Plan	Adhoc	SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'
2	Compiled Plan	Prepared	(@0 varchar(8000))select * from BigOrders where CustomerID = @0

In this example, forcing SQL Server to treat the constant as a parameter is not a good thing, and the batch sets the database back to SIMPLE PARAMETERIZATION (the default) as the last step. Note also that while we are using PARAMETERIZATION FORCED, the datatype chosen for the autoparameterized query is the largest possible regular character datatype.

So what can you do if you have many queries that should not be autoparameterized, and many others that should be? As we've seen, the SQL Server query processor is much more conservative about deciding whether a template is safe than an application can be. SQL Server guesses which values are really parameters, whereas your application should actually know. Rather than rely on autoparameterization in all cases, you can use one of the prepared query mechanisms to mark values as parameters when they are known.

The SQL Server Performance Monitor includes a counter called SQLServer:SQL Statistics that has several counters dealing with autoparameterization. You can monitor these counters to determine whether there are many unsafe or failed autoparameterization attempts. If these numbers are high, you can inspect your

applications for situations in which the application can take responsibility for explicitly marking the parameters.

Prepared Queries

As we saw previously, a query that is autoparameterized by SQL Server shows an objtype of prepared in the cached plan metadata. There are two other constructs that also show up as having prepared plans. Both of these constructs allow the programmer to take control over which values are parameters and which aren't and in addition, unlike simple autoparameterization, the programmer also determines the datatype that will be used for the parameters. One construct is the SQL Server stored procedure `sp_executesql`, that is called from within a Transact-SQL batch, and the other is to use the prepare/execute method from the client application.

The Procedure

The stored procedure `sp_executesql` is halfway between adhoc caching and stored procedures. Using `sp_executesql` requires that you identify the parameters and their datatypes, but doesn't require all the persistent object management needed for stored procedures and other programmed objects.

Here's the general syntax for the procedure:

```
sp_executesql @batch_text, @batch_parameter_definitions,
               param1,...paramN
```

Repeated calls with the same values for `@batch_text` and `@batch_parameter_definitions` use the same cached plan, with the new parameter values specified. The plan will be reused as long as the plan has not been removed from cache, and as long as it is still valid. The section "[Causes of Recompilation](#)," later in this chapter, discusses those situations in which SQL Server determines that a plan is no longer valid. The same cached plan can be used for all the following queries:

```
EXEC sp_executesql N'SELECT FirstName, LastName, Title
                     FROM Employees
                     WHERE EmployeeID = @p', N'@p tinyint', 6
EXEC sp_executesql N'SELECT FirstName, LastName, Title
                     FROM Employees
                     WHERE EmployeeID = @p', N'@p tinyint', 2
EXEC sp_executesql N'SELECT FirstName, LastName, Title
                     FROM Employees
                     WHERE EmployeeID = @p', N'@p tinyint', 6
```

Just like forcing autoparameterization, using `sp_executesql` to force reuse of a plan is not always appropriate. If we take the same example used earlier when we set the database to PARAMETERIZATION FORCED, we can see that using `sp_executesql` is just as inappropriate.

```
USE Northwind2;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
                     WHERE CustomerID = @p', N'@p nvarchar(10)', 'CENTC';
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
```

```

WHERE CustomerID = @p', N'@p nvarchar(10)', 'SAVEA';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SET STATISTICS IO OFF;
GO

```

Again, we can see that the first SELECT required 12 logical reads and the second required 312. The output of the usecount query shows the parameterized query being used twice. Note that with sp_executesql, we do not have any entries for the adhoc shell query (unparameterized) queries.

Usecounts	Cacheobjtype	Objtype	Text
2	Compiled Plan	Prepared	(@p nvarchar(10))SELECT * FROM BigOrders WHERE CustomerID = @p

The Prepare and Execute Method

This last mechanism is like sp_executesql in that parameters to the batch are identified by the application, but there are some key differences. The prepare and execute method does not require the full text of the batch to be sent at each execution. Rather, the full text is sent once at prepare time; a handle that can be used to invoke the batch at execute time is returned. ODBC and OLE DB expose this functionality via SQLPrepare/SQLEExecute and ICommandPrepare. You can also use this mechanism via ODBC and OLE DB when cursors are involved. When you use these functions, SQL Server is informed that this batch is meant to be used repeatedly.

Caching Prepared Queries

If your queries have been parameterized at the client using the Prepare/Execute method, the metadata will show you prepared queries, just as for queries that are parameterized at the server, either automatically or by using sp_executesql. However, queries that are not autoparameterized (either under SIMPLE or FORCED parameterization) will not have any corresponding adhoc shell queries in cache, containing the unparameterized actual values; they will only have the prepared plans. There is no guaranteed way to detect whether a prepared plan was prepared through autoparameterization or through client-side parameterization. If you see a corresponding shell query, you can know that the query was autoparameterized, but the opposite is not always true. Because the shell queries have a zero cost, they are among the first candidates to be removed when SQL Server is under memory pressure. So a lack of a shell query might just mean that adhoc plan was already removed from cache, not that there never was a shell query.

Compiled Objects

When looking at the metadata in sys.dm_exec_cached_plans, we've seen compiled plans with objtype values of adhoc and prepared. The third objtype value that we will be discussing is Proc, and you will see this type used when executing stored procedures, triggers, user-defined scalar functions, and multistatement table-valued functions. With the exception of triggers, which are never called directly, you have full control over what values are parameters and what their datatypes are when executing these objects.

Stored Procedures

Stored procedures and user-defined scalar functions are treated almost identically. The metadata indicates that a compiled plan with an objtype value of Proc is cached and can be reused repeatedly. By default, the cached plan will be reused for all successive executions, and as we've seen with the sp_executesql, this is not always desirable. However, unlike the plans cached and reused with sp_executesql, you have an option with stored procedures and user-defined scalar functions to force recompilation when the object is executed. In addition, for stored procedures, you can create the object so that a new plan is created every single time it is executed.

To force recompilation for a single execution, you can use the EXECUTE . . . WITH RECOMPILE option. Here is an example in the Northwind2 database of forcing recompilation for a stored procedure:

```
USE Northwind2;
GO
CREATE PROCEDURE P_Customers
    @cust nvarchar(10)
AS
    SELECT RowNum, CustomerID, OrderDate, ShipCountry
    FROM BigOrders
    WHERE CustomerID = @cust;
GO
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
GO
EXEC P_Customers 'CENTC';
GO
EXEC P_Customers 'SAVEA';
GO
EXEC P_Customers 'SAVEA' WITH RECOMPILE;
```

If you look at the output from STATISTICS IO, you'll see that the second execution used a suboptimal plan that required more pages to be read than would be needed by a table scan. This is the situation that you may have seen referred to as parameter sniffing. SQL Server is basing the plan for the procedure on the first actual parameter, in this case, CENTC, and then subsequent executions assume the same or a similar parameter is used. The third execution uses the WITH RECOMPILE option to force SQL Server to come up with a new plan, and you should see that the number of logical page reads is equal to the number of pages in the table.

If you look at the results from running the usecounts query, you should see that the cached plan for P_Customers procedure has a usecounts value of 2, instead of 3. The plan developed for a procedure executed with the WITH RECOMPILE option is just considered valid for the current execution, and is never kept in cache for reuse.

Usecounts	Cacheobjtype	Objtype	Text
2	Compiled Plan	Proc	CREATE PROCEDURE P_Customers @cust nvarchar(10) AS SELECT RowNum, CustomerID, OrderDate, ShipCountry FROM BigOrders WHERE CustomerID = @cust

Functions

User-defined scalar functions can behave exactly the same way as procedures. If you execute them using the EXECUTE statement, instead of as part of an expression, you can also force recompilation. Here is an example of a function that masks part of a Social Security number. We will create it in the pubs sample database, because the authors table contains a Social Security number in the au_id column.

Code View:

```
USE pubs;
GO
CREATE FUNCTION dbo.fnMaskSSN (@ssn char(11))
RETURNS char(11)
AS
BEGIN
    SELECT @SSN = 'xxx-xx-' + right (@ssn, 4)
    RETURN @SSN
END;
GO
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
GO

DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-45-6789';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111' WITH RECOMPILE;
SELECT @mask;
GO
```

If you run the usecounts query you should notice the cached plan for the function has an objtype of proc and has a usecounts value of 2. If a scalar function is used within an expression, as in the example below, there is no way to request recompilation.

```
SELECT dbo.fnMaskSSN(au_id), au_lname, au_fname, au_id FROM authors;
```

TVFs may or may not be treated like procedures depending on how you define them. You can define a TVF as an inline function, or as a multistatement function. Neither method allows you to force recompilation when the function is called. Here are two functions that do the same thing:

```
CREATE FUNCTION Fnc_Inline_Customers (@cust nvarchar(10))
RETURNS TABLE
AS
    RETURN
        (SELECT RowNum, CustomerID, OrderDate, ShipCountry
        FROM BigOrders
        WHERE CustomerID = @cust);
GO

CREATE FUNCTION Fnc_Multi_Customers (@cust nvarchar(10))
```

```

RETURNS @T TABLE (RowNum int, CustomerID nchar(10), OrderDate datetime,
                  ShipCountry nvarchar(30))
AS
BEGIN
    INSERT INTO @T
        SELECT RowNum, CustomerID, OrderDate, ShipCountry
        FROM BigOrders
        WHERE CustomerID = @cust
    RETURN
END;
GO

```

Here are the calls to the functions:

```

DBCC FREEPROCCACHE
GO
SELECT * FROM Fnc_Multi_Customers('CENTC')
GO
SELECT * FROM Fnc_Inline_Customers('CENTC')
GO
SELECT * FROM Fnc_Multi_Customers('SAVEA')
GO
SELECT * FROM Fnc_Inline_Customers('SAVEA')
GO

```

If you run the usecounts query you will see that only the multistatement function has its plan reused. The inline function is actually treated like a view, and the only way the plan can be reused would be if the exact same query were reexecuted, that is, if the same SELECT statement called the function with the exact same parameter.

Causes of Recompilation

Up to this point, we've been discussing the situations in which SQL Server automatically reuses a plan, and the situation in which a plan may be reused inappropriately so that you need to force recompilation. However, there are also situations in which an existing plan will not be reused because of changes to the underlying objects or the execution environment. The reasons for these unexpected recompliations fall into one of two different categories, which we call correctness-based recompiles and optimality-based recompiles.

Correctness-Based Recompiles

SQL Server may choose to recompile a plan if it has a reason to suspect that the existing plan may no longer be correct. This can happen when there are explicit changes to the underlying objects, such as changing a datatype or dropping an index. Obviously, any existing plan that referenced the column assuming its former datatype, or that accessed data using the now nonexistent index, would not be correct. Correctness-based recompiles fall into two general categories: schema changes and environmental changes. The following changes mark an object's schema as changed:

- Adding or dropping columns to/from a table or view.
- Adding or dropping constraints, defaults, or rules to or from a table.
- Adding an index to a table or an indexed view.
- Dropping an index defined on a table or an indexed view if the index is used by the plan.
- Dropping a statistic defined on a table will cause a correctness-related recompilation of any query plans that use that table.

- Adding or dropping a trigger from a table.

In addition, running the procedure sp_recompile on a table or view will change the modification date for the object. Which you can observe in the modify_date column in sys.objects. This will make SQL Server believe that a schema change has occurred so that recompilation will take place at the next execution of any stored procedure, function, or trigger that accesses the table or view. Running sp_recompile on a procedure, trigger, or function will clear all the plans for the executable object out of cache, to guarantee that the next time it is executed, it will be recompiled.

Other correctness-based recompiles are invoked when the environment changes by changing one of a list of SET options. Changes in certain SET options can cause a query to return different results, so when one of these values changes, SQL Server wants to make sure a plan is used that was created in a similar environment. SQL Server keeps track of which SET options are set when a plan is executed, and you have access to a bitmap of these SET options in the DMF called sys.dm_exec_plan_attributes. This function is called by passing in a plan handle value that you can obtain from the sys.dm_exec_cached_plans view and returns one row for each of a list of plan attributes. You'll need to make sure you include plan_handle in the list of columns to be retrieved, not just the few columns we used earlier in the usecounts query. Here's an example of retrieving all the plan attributes when we supply a plan_handle value. **Table 5-1** shows the results.

```
SELECT * FROM sys.dm_exec_plan_attributes
(0x06001200CF0B831CB821AA05000000000000000000000000000000)
```

Table 5-1. Attributes Corresponding to a Particular

AttributeValue is_cache_keyset_options43471objectid4783502871dbid181dbid_execute01user_id-21language_id01date_form

Later in the chapter, when we explore cache management and caching internals, you'll learn about some of these values in which the meaning is not obvious. To get the attributes to be returned in a row along with each plan_handle, you can use the SQL Server 2005 PIVOT operator, and list each of the attributes that you want to turn into a column. In this next query, we want to retrieve the set_options, the object_id, and the sql_handle from the list of attributes.

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
      FROM sys.dm_exec_cached_plans
      OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
      WHERE cacheobjtype = 'Compiled Plan'
    ) AS ecpa
PIVOT (MAX(epca.value) FOR epca.attribute
      IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

We get a value of 4,347 for set_options which is equivalent to the bit string 1000011111011. To see which bit refers to which SET options, we could change one option and then see how the bits have changed. For example, if we clear the plan cache and change ANSI_NULLS to OFF, the set_options value will change to 4315, or binary 1000011011011. The difference is the 6th bit from the right, which has a value of 32, the difference between 4,347 and 4,315. If we didn't clear the plan cache, we would end up with two plans for the same batch, one for each set_options value.

Not all changes to SET options will cause a recompile, although many of them will. The following is a list of the SET options that will cause a recompile when changed:

- ANSI_NULL_DFLT_OFF
- ANSI_NULL_DFLT_ON
- ANSI_NULLS
- ANSI_PADDING
- ANSI_WARNINGS
- ARITHABORT
- CONCAT_NULL_YIELDS_NULL
- DATEFIRST
- DATEFORMAT
- FORCEPLAN
- LANGUAGE
- NO_BROWSETABLE
- NUMERIC_ROUNDABORT
- QUOTED_IDENTIFIER

Two of the SET options in the list above have a special behavior in relationship to objects, including stored procedures, functions, views, and triggers. The SET option settings for ANSI_NULLS and QUOTED_IDENTIFIER are actually saved along with the object definition and the procedure or function will always execute with the SET values as they were when the object was first created. You can determine what values these two SET options had for your objects by selecting from the OBJECTPROPERTY function, as shown:

```
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsQuotedIdentOn');
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsAnsiNullsOn');
```

A returned value of 0 means the SET option is OFF, a value of 1 means the option is ON, and a value of NULL means that you typed something incorrectly or that you don't have appropriate permissions. However, even though changing the value of either of these options does not cause any difference in execution of the objects, SQL Server may still recompile the statement that accesses the object. The only objects for which RECOMPILE will be avoided is for cached plans with an objtype value of Proc, namely stored procedures, multistatement TVFs, and triggers. For these compiled objects, the usecounts query will show you the same plan being reused, and will not show additional plans with different set_options values. Inline TVFs and views will create new plans if these options are changed, and the set_options value will indicate a different bitmap. However, the behavior of the underlying SELECT statement will not change.

Optimality-Based Recompiles

SQL Server may also choose to recompile a plan if it has reason to suspect that the existing plan is no longer optimal. The primary reasons for suspecting a nonoptimal plan deal with changes to the underlying data. If any of the statistics used to generate the query plan have been updated since the plan was created, or if any of the statistics are considered stale, SQL Server will recompile the query plan.

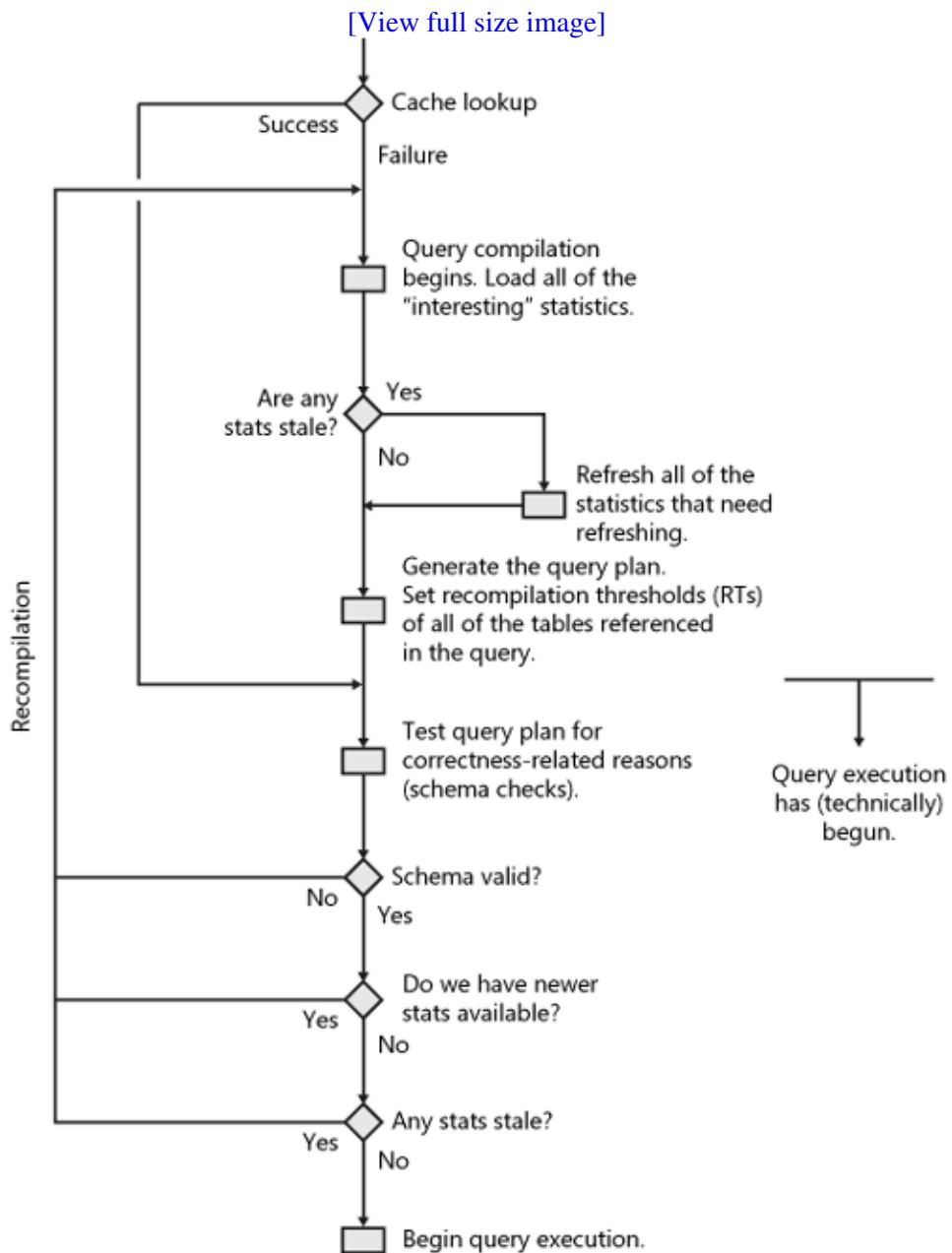
Updated Statistics

Statistics can be updated either manually or automatically. Manual updates happen when someone runs sp_updatestats or the UPDATE STATISTICS command. Automatic updates happen when SQL Server determines that existing statistics are out of date, or stale, and these updates only happen when the database has the option AUTO_UPDATE_STATISTICS set to ON. This could happen if another batch had tried to use one of the same tables or indexes used in the current plan, detected the statistics were stale, and initiated an UPDATE STATISTICS operation.

Stale Statistics

SQL Server will detect out-of-date statistics when it is first compiling a batch that has no plan in cache. It will also detect stale statistics for existing plans. [Figure 5-1](#) shows a flowchart of the steps involved in finding an existing plan and checking to see if recompilation is required. You can see that SQL Server checks for stale statistics after first checking to see if there already are updated statistics available. If there are stale statistics, the statistics will be updated, and then a recompile will begin on the batch. If `AUTO_UPDATE_STATISTICS_ASYNC` is ON for the database, SQL Server will not wait for the update of statistics to complete, it will just recompile based on the stale statistics.

Figure 5-1. Checking an existing plan to see if recompilation is necessary



Statistics are considered to be stale if a sufficient number of modifications have occurred on the column supporting the statistics. Each table has a recompilation threshold, or RT, that determines how many changes can take place before any statistics on that table are marked as stale. The RT values for all of the tables

referenced in a batch are stored with the query plans of that batch.

The RT values depend on the type of table, that is, whether it is permanent or temporary, and on the current number of rows in the table at the time a plan is compiled. The exact algorithms for determining the RT values are subject to change with each service pack, so I will show you the algorithm for the RTM release of SQL Server 2005. The formulas used in the various service packs will be similar to this, but are not guaranteed to be exactly the same. N indicates the cardinality of the table.

- For both permanent and temporary tables, if N is less or equal to 500, the RT value is 500. This means that for a relatively small table, you must make at least 500 changes to trigger recompilation. For larger tables, at least 500 changes must be made, plus 20 percent of the number of rows.
- For temporary tables, the algorithm is the same, with one exception. If the table is very small or empty (N is less than 6 prior to any data modification operations), all we need are 6 changes to trigger a recompile. This means that a procedure that creates a temporary table, which is empty when created, and then inserts 6 or more rows into that table, will have to be recompiled as soon as the temp table is accessed.
- You can get around this frequent recompilation of batches that create temporary tables by using the KEEP PLAN query hint. Use of this hint changes the recompilation thresholds for temporary tables and makes them identical to those for permanent tables. So if changes to temporary tables are causing many recompilations, and you suspect that the recompilations are affecting overall system performance, you can use this hint and see if there is a performance improvement. The hint can be specified as shown in this query:

```
SELECT <column list>
FROM dbo.PermTable A INNER JOIN #TempTable B ON A.col1 = B.col2
WHERE <filter conditions>
OPTION (KEEP PLAN)
```

For table variables, there is no RT value. This means that you will not get recompilations caused by changes in the number of rows in a table variable.

Modification Counters

The RT values discussed above are the number of changes required for SQL Server to recognize that statistics are stale. In versions of SQL Server prior to SQL Server 2005, the sysindexes system table keeps track of the number of changes that had actually occurred in a table in a column called rowmodctr. These counters keep track of any changes in any row of the table or index, even if the change was to a column that was not involved in any index or useful statistics. SQL Server 2005 now uses a set of Column Modification Counters or colmodctr values, with a separate count being maintained for each column in a table, except for computed nonpersisted columns. These counters are not transactional, which means that if a transaction starts, inserts thousands of rows into a table, and then is rolled back, the changes to the modification counters will not be rolled back. Unlike the rowmodctr values in sysindexes, the colmodctr values are not visible to the user. They are only available internally to the query processor.

Tracking Changes to Tables and Indexed Views Using Values

The colmodctr values that SQL Server keeps track of are continually modified as the table data changes. [Table 5-2](#) describes when and how the colmodctr values are modified based on changes to your data, including INSERT, UPDATE, DELETE, bulk insert, and TRUNCATE TABLE operations. Although we are only mentioning table modifications specifically, keep in mind the same colmodctr values are kept track of for indexed views.

Table 5-2. Factors Affecting Changes to the Internal Values

StatementChanges to colmodctr Values

INSERT	All colmodctr values increased by 1 for each row inserted
DELETE	All colmodctr values increased by 1 for each row deleted
UPDATE	If the update is to nonkey columns: colmodctr values for modified columns are increased by 1 for each row updated. If the update is to key columns: colmodctr values are increased by 2 for all of the columns in the table, for each row updated.
Bulk insert	Treated like N INSERT operations. All colmodctr values increased by N where N is the number of rows bulk inserted.
Table truncation	Treated like N DELETE operations. All colmodctr values increased by N where N is the table's cardinality.

Skipping the Recompilation Step

There are several situations in which SQL Server will bypass recompiling a statement for plan optimality reasons. These include:

- When the plan is a trivial plan. A trivial plan is one for which there are no alternative plans, based on the tables referenced by the query, and the indexes (or lack of indexes) on those tables. In these cases, where there really is only one way to process a query, any recompilation would be a waste of resources, no matter how much the statistics had changed. Keep in mind that there is no assurance that a query will continue to have a trivial plan just because it originally had a trivial plan. If new indexes have been added since the query was last compiled, there may now be multiple possible ways to process the query.
- If the query contains the OPTION hint KEEPFIXED PLAN, SQL Server will not recompile the plan for any optimality-related reasons.
- If automatic updates of statistics for indexes and statistics defined on a table or indexed view are disabled, all plan optimality-related recompliations caused by those indexes or statistics will stop.

Note



Turning off the auto-statistics feature is usually not a good idea because the query optimizer will no longer be sensitive to data changes in those objects, and suboptimal query plans could easily result. You can consider using this technique only as a last resort after exhausting all of the other alternative ways to avoid recompilation. Make sure you thoroughly test your applications after changing the auto-statistics options to verify that you are not hurting performance in other areas.

- If all of the tables referenced in the query are read-only, SQL Server will not recompile the plan.

Multiple Recompilations

In the previous discussion of unplanned recompilation, we primarily described situations in which a cached plan would be recompiled prior to execution. However, even if SQL Server decides it can reuse an existing plan, there may be cases where stale statistics or schema changes are discovered after the batch begins execution, and then a recompile will occur after execution starts. Each batch or stored procedure can contain multiple query plans, one for each optimizable statement. Before SQL Server begins executing any of the individual query plans, it checks for correctness and optimality of that plan. If one of the checks fails, the corresponding statement is compiled again, and a possibly different query plan is produced.

In some cases, query plans may be recompiled even if the plan for the batch was not cached. For example, if a batch contains a literal larger than 8 KB, it will never be cached. However, if this batch creates a temporary table, and then inserts multiple rows into that table, the insertion of the seventh row will cause a recompilation

because of passing the recompilation threshold for temporary tables. Because of the large literal, the batch was not cached, but the currently executing plan needs to be recompiled.

In SQL Server 2000, when a batch was recompiled, all of the statements in the batch were recompiled, not just the one that initiated the recompilation. SQL Server 2005 introduces statement-level recompilation, which means that only the statement that causes the recompilation has a new plan created, not the entire batch. This means that SQL Server 2005 will spend less CPU time and memory during recompilations.

Removing Plans from Cache

In addition to needing to recompile a plan based on schema or statistics changes, SQL Server will need to compile plans for batches if all previous plans have been removed from the plan cache. Plans are removed from cache based on memory pressure, which we'll talk about in the section on "Cache Management." However, other operations can cause plans to be removed from cache. Some of these operations remove all the plans from a particular database, and others remove all the plans for the entire SQL Server instance.

The following operations flush the entire plan cache so that all batches submitted afterwards will need a fresh plan. Note that although some of these operations only affect a single database, the entire plan cache is cleared.

- Detaching any database.
- Upgrading any database to SQL Server 2005 (on SQL Server 2005 server).
- Running the DBCC FREEPROCCACHE command.
- Running the ALTER DATABASE . . . MODIFY FILEGROUP command for any database.
- Modifying a collation for any database using ALTER DATABASE . . . COLLATE command.

The following operations clear all plans associated with a particular database and cause new compilations the next time a batch is executed.

- Running the DBCC FLUSHPROCINDB command.
- Altering a database with any of the following options:
 1. ALTER DATABASE . . . MODIFY NAME=command
 2. ALTER DATABASE . . . SET ONLINE command
 3. ALTER DATABASE . . . SET OFFLINE command
 4. ALTER DATABASE . . . SET EMERGENCY command
 5. Dropping a DATABASE
 6. When a database auto-closes

Note that there is no way to force SQL Server to remove just a single query plan from cache.

Plan Cache Internals

Knowing when and how plans are reused or recompiled can help you design well-performing applications. The more you understand about optimal query plans, and how different actual values and cardinalities require different plans, the more you can determine when recompilation is a useful thing. When you are getting unnecessary recompiles, or when SQL Server is not recompiling when you think it should, your troubleshooting efforts will be easier the more you know about how plans are managed internally. In this section, we'll explore the internal organization of the plan cache, the metadata available, how SQL Server finds a plan in cache, plan cache sizing, and SQL Server's plan eviction policy.

Cache Stores

SQL Server's plan cache is made up of four separate memory areas, called cache stores. There are actually other stores in SQL Server's memory, which can be seen in the Dynamic Management View (DMV) called sys.dm_os_memory_cache_counters, but there are only four that contain query plans. The names in parentheses below are the values that can be seen in the type column of sys.dm_os_memory_cache_counters:

- Object Plans (CACHESTORE_OBJCP)

Object Plans include plans for stored procedures, functions, and triggers

- SQL Plans (CACHESTORE_SQLCP)

SQL Plans include the plans for adhoc cached plans, autoparameterized plans, and prepared plans.

- Bound Trees (CACHESTORE_PHDR)

Bound Trees are the structures produced by SQL Server's algebrizer for views, constraints, and defaults.

- Extended Stored Procedures (CACHESTORE_XPROC)

Extended Procs (Xprocs) are predefined system procedures, like sp_executesql and sp_tracecreate, that are defined using a DLL, not using Transact-SQL statements. The cached structure contains only the function name and the DLL name in which the procedure is implemented.

Each plan cache store contains a hash table to keep track of all the plans in that particular store. Each bucket in the hash table contains zero, one, or more cached plans. When determining which bucket to use, SQL Server uses a very straightforward hash algorithm. The hash key is computed as (object_id * database_id) mod (hash table size). For plans that are associated with adhoc or prepared plans, the object_id is an internal hash of the batch text. The DMV sys.dm_os_memory_cache_hash_tables contains information about each hash table, including its size. You can query this view to retrieve the number of buckets for each of the plan cache stores using the following query:

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP',
'CACHESTORE_PHDR', 'CACHESTORE_XPROC');
```

You should notice that the Bound Trees store is about 10 percent of the size of the stores for Object Plans and SQL Plans. The size of the store for Extended Stored Procedures is always set to 127 entries. We will not be discussing Bound Trees and Extended Stored Procedures further. The rest of the chapter dealing with caching of plans will be concerned only with Object Plans and SQL Plans.

Finding a plan in cache is a two-step process. The hash key described previously leads SQL Server to the bucket in which a plan might be found, but if there are multiple entries in the bucket, SQL Server needs more information to determine if the exact plan it is looking for can be found. For this second step, it needs a cache key, which is a combination of several attributes of the plan. Earlier, we looked at the DMF sys.dm_exec_plan_attributes, to which we could pass a plan_handle. The results obtained were a list of attributes for a particular plan, and a Boolean value indicating whether that particular value was a cache key. Table 5-1 included 11 attributes that comprise the cache key, and SQL Server needs to make sure all 11 values match before determining that it has found a matching plan in cache.

Compiled Plans

There are two main types of plans in the Object and SQL plan cache stores: compiled plans and execution plans. Compiled plans are the type of object we have been looking at up to this point, when examining the sys.dm_exec_cached_plans view. We have already discussed the three main objtype values that can correspond to a compiled plan: adhoc, prepared, and proc. Compiled plans can be stored in either the Object store or the SQL store depending on which of those three objtype values they have. The compiled plans are considered valuable memory objects, since they can be costly to re-create. SQL Server will attempt to keep them in cache. When SQL Server experiences heavy memory pressure, the policies used to remove cache objects ensure that our compiled plans are not the first objects to be removed.

A compiled plan is generated for an entire batch, not just for a single statement. For a multistatement batch, you can think of the compiled plan as an array of plans, with each element of the array containing a query plan for an individual statement. Compiled plans can be shared between multiple sessions or users. However, you should be aware that not every user executing the same plan will get the same results, even if there is no change to the underlying data. Unless the compiled plan is an adhoc plan, each user will have their own parameters, their own local variables, and the batch may build temporary tables or worktables specific to that user. The information specific to one particular execution of a compiled plan is stored in another structure called the executable plan.

Execution Plans

Executable plans, or execution contexts, are considered to be dependent on compiled plans and do not show up in the sys.dm_exec_cached_plans view. Executable plans are runtime objects created when a compiled plan is executed. Just as for compiled plans, executable plans can be object plans stored in the object store, or SQL plans, stored in the SQL store. Each executable plan exists in the same cache store as the compiled plan to which it is dependent. Executable plans contain the particular runtime information for one execution of a compiled plan, and include the actual runtime parameters, any local variable information, object ids for objects created at run time, the user ID, and information about the currently executing statement in the batch.

When SQL Server starts executing a compiled plan, it generates an executable plan from that compiled plan. Each individual statement in a compiled plan gets its own executable plan, which you can think of as a runtime query plan. Unlike compiled plans, executable plans are for a single session. For example, if there are 100 users simultaneously executing the same batch, there will be 100 executable plans for the same compiled plan. Executable plans can be regenerated from their associated compiled plan, and they are relatively inexpensive to create. Later in this section, we'll look at the sys.dm_exec_plan_dependent_objects view, which contains information about your executable plans.

Plan Cache Metadata

We have already looked at some of the information in the DMV sys.dm_exec_cached_plans when we looked at usecount information to determine whether or not our plans were being reused. In this section, we'll look at some of the other metadata objects and discuss the meaning of some of the data contained in the metadata.

Handles

The sys.dm_exec_cached_plans view contains a value called a plan_handle for every compiled plan. The plan_handle is a hash value that SQL Server derives from the compiled plan of the entire batch, and it is guaranteed to be unique for every currently existing compiled plan. (The plan_handle values can be reused over time.) The plan_handle can be used as an identifier for a compiled plan. The plan_handle remains the same even if individual statements in the batch are recompiled because of the correctness or optimality

reasons discussed earlier.

As mentioned, the compiled plans are stored in the two cache stores, depending on whether the plan is an object plan or a SQL plan. The actual SQL Text of the batch or object is stored in another cache called the SQL Manager Cache (SQLMGR). The Transact-SQL Text associated with each batch is stored in its entirety, including all the comments. The Transact-SQL Text cached in the SQLMGR cache can be retrieved using a data value called the sql_handle. The sql_handle contains a hash of the entire batch text, and because it is unique for every batch, the sql_handle can serve as an identifier for the batch text in the SQLMGR cache.

Any specific Transact-SQL batch will always have the same sql_handle, but it may not always have the same plan_handle. If the cache keys change, we'll get a new plan_handle in plan cache. Refer back to [Table 5-1](#) to see which plan attributes make up the cache keys. The relationship between sql_handle and plan_handle is thus 1:N.

We've seen that plan_handle values can be easily obtained from the view sys.dm_exec_cached_plans. We can get the sql_handle value that corresponds to a particular plan_handle from the sys.dm_exec_plan_attributes function that we looked at earlier. Here is the same query we discussed earlier to return attribute information and pivot it so that three of the attributes are returned in the same row as the plan_handle value.

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
      FROM sys.dm_exec_cached_plans
      OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
     WHERE cacheobjtype = 'Compiled Plan'
   ) AS ecpa
PIVOT (MAX(epca.value) FOR ecpa.attribute
      IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

The view sys.dm_exec_query_stats contains both plan_handle and sql_handle values, as well as information about how often each plan was executed and how much work was involved in the execution. However, because plan_handle is a very cryptic value, it's difficult to determine which of our queries each sql_handle corresponds to. To get that information, we can use another function.

sys.dm_exec_sql_text

The function sys.dm_exec_sql_text can take either a sql_handle or a plan_handle as a parameter, and it will return the SQL Text that corresponds to the handle. Any sensitive information like passwords, that might be contained in the SQL Text, are blocked when the SQL is returned. The text column in the functions output contains the entire SQL batch text for adhoc, prepared, and autoparameterized queries, and for objects like triggers, procedures, and functions, it gives the full object definition.

Viewing the SQL Text from sys.dm_exec_sql_text is useful in quickly identifying identical batches that may have different compiled plans because of several factors, like SET option differences. As an example, consider the code below, which executes two identical batches. The only difference between the two consecutive executions is that the value of the SET option QUOTED_IDENTIFIER has changed. It is OFF in the first execution and ON in the second. After executing both batches, we examine the sys.dm_exec_query_stats view.

Code View:

```
USE Northwind2;
DBCC FREEPROCCACHE;
SET QUOTED_IDENTIFIER OFF;
GO
--- this is an example of the relationship between
```

```
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
SET QUOTED_IDENTIFIER ON;
GO
--- this is an example of the relationship between
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
SELECT st.text, qs.sql_handle, qs.plan_handle
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(sql_handle) st;
GO
```

You should see two rows with the same text string and sql_handle, but with different plan_handle values, as shown below. (In our output, the difference between the two plan_handle values is only a single digit so it may be hard to see, but in other cases the difference may be more obvious.)

Text	sql_handle	plan_handle
--- this is an example of the relationship between sql_handles and plan_handles interaction SELECT LastName, FirstName, Coun- try FROM Employees WHERE Country <> 'USA'	0x0200000012330B0EE A82077439354E7A5B12 E1B7E37A1361	0x0600120012330B0EB 82187050000000000000 0000000000000000
--- this is an example of the relationship between sql_handles and plan_handles interaction SELECT LastName, FirstName, Coun- try FROM Employees WHERE Country <> 'USA'	0x0200000012330B0EE A82077439354E7A5B12 E1B7E37A1361	0x0600120012330B0EB 82186050000000000000 0000000000000000

We can see that we have two plans corresponding to the same batch text, and this example should make clear the importance of making sure that all the SET options that affect plan caching should be the same when the same queries are executed repeatedly. You should verify whatever changes your programming interface makes to your SET options to make sure you don't end up with different plans unintentionally. Not all interfaces use the same defaults for the SET option values. For example, the OSQL interface uses the ODBC driver, which sets QUOTED_IDENTIFIER to OFF for every connection, while SQL Server Management Studio uses ADO.NET, which sets QUOTED_IDENTIFIER to ON. Executing the same batches from these two different clients will result in multiple plans in cache.

sys.dm_exec_cached_plans

The sys.dm_exec_cached_plans view is the one we use most often for troubleshooting query plan recompilation issues. It's the one we used in the first section to illustrate the plan reuse behavior of adhoc plans compared to autoparameterized and prepared plans. This view has one row per cached plan, and in

addition to the plan_handle and usecounts, which we've looked at already, this DMV has other useful information about the cached plans, including:

- size_in_bytes: number of bytes consumed by this cache object.
- cacheobjtype: type of the cache object, that is, if it's a compiled plan, or a Parse Tree or an Extended Proc.
- memory_object_address: memory address of the cache object, which can be used to get the memory breakdown of the cache object.

Although this DMV does not have the SQL Text associated with each compiled plan, we've seen that we can find it by passing the plan_handle to the sys.dm_exec_sql_text function. We can use the query below to retrieve the text, usecounts, and size_in_bytes of the compiled plan and cacheobjtype for all the plans in cache. The results will be returned in order of frequency, with the batch having the most use showing up first:

```
SELECT st.text, cp.plan_handle, cp.usecounts, cp.size_in_bytes,
       cp.cacheobjtype, cp.objtype
  FROM sys.dm_exec_cached_plans cp
    CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
 ORDER BY cp.usecounts DESC
```

sys.dm_exec_cached_plan_dependent_objects

This DMF returns one row for every dependent object of a compiled plan when you pass a valid plan_handle in as a parameter. If the plan_handle is not that of a compiled plan, the function will return NULL. Dependent objects include executable plans, as discussed previously, as well as plans for cursors used by the compiled plan. The example below uses sys.dm_exec_cached_plan_dependent_objects, as well as sys.dm_exec_cached_plans, to retrieve the dependent objects for all compiled plans, the plan_handle, and their usecounts. It also calls the sys.dm_exec_sql_text function to return the associated Transact-SQL batch.

```
SELECT text, plan_handle, d.usecounts, d.cacheobjtype
  FROM sys.dm_exec_cached_plans
    CROSS APPLY sys.dm_exec_sql_text(plan_handle)
    CROSS APPLY
      sys.dm_exec_cached_plan_dependent_objects(plan_handle) d;
```

sys.dm_exec_requests

The sys.dm_exec_requests view returns one row for every currently executing request within your SQL Server instance and is useful for many purposes in addition to tracking down plan cache information. This DMV contains the sql_handle and the plan_handle for the current statement, as well as resource usage information for each request. For troubleshooting purposes you can use this view to help identify long-running queries. Keep in mind that the sql_handle points to the Transact-SQL for the entire batch. However, the sys.dm_exec_requests view contains columns statement_start_offset and statement_end_offset that indicate the position within the entire batch where the currently executing statement can be found. The offsets start at 0 and an offset of -1 indicates the end of the batch. The statement offsets can be used in combination with the sql_handle passed to sys.dm_exec_sql_text to extract the query text from the entire batch text as demonstrated next. This query returns the 10 longest-running queries currently executing:

```
SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
   ((CASE statement_end_offset
      WHEN -1
```

```

        THEN DATALENGTH(text)
    ELSE statement_end_offset
END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_requests
    CROSS APPLY sys.dm_exec_sql_text(sql_handle)
ORDER BY total_elapsed_time DESC

```

Note that including the '*' in the SELECT list indicates that this query should return all the columns from sys.dm_exec_requests view. You should replace the '*' with the columns that you are particularly interested in, such as start_time, blocking_session_id, etc.

sys.dm_exec_query_stats

Just as the text returned from the sql_handle is the text for the entire batch, the compiled plans that are returned are for the entire batch. For optimum troubleshooting, we can use sys.dm_exec_query_stats to return performance information for individual queries within a batch. This view returns performance statistics for queries, aggregated across all executions of the same query. This view also returns both a sql_handle and a plan_handle, as well as the start and end offsets like we saw in sys.dm_exec_requests. The following query will return the top-10 queries by total CPU time, to help you identify the most expensive queries running on your SQL Server.

```

SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
    ((CASE statement_end_offset
        WHEN -1
            THEN DATALENGTH(text)
        ELSE statement_end_offset
    END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_query_stats
    CROSS APPLY sys.dm_exec_sql_text(sql_handle)
    CROSS APPLY sys.dm_exec_query_plan(plan_handle)
ORDER BY total_elapsed_time/execution_count DESC;

```

The sys.dm_exec_query_stats returns a tremendous amount of performance information for each query, including the number of times it was executed, and the cumulative I/O, CPU, and duration information. Keep in mind that this view is only updated when a query is completed, so you might need to retrieve information multiple times if there is currently a large workload on your server.

Cache Size Management

We've already talked about plan reuse and how SQL Server finds a plan in cache. In this section, we'll look at how SQL Server manages the size of plan cache, and how it determines which plans to remove if there is no room left in cache. Earlier, I discussed a few situations in which plans would be removed from cache. These situations included global operations like running DBCC FREEPROCCACHE to clear all plans from cache, as well as changes to a single procedure, such as ALTER PROCEDURE, which would drop all plans for that procedure from cache. In most other situations, plans will only be removed from cache when memory pressure is detected. The algorithm that SQL Server uses to determine when and how plans should be removed from cache is called the eviction policy. Each cache store can have its own eviction policy, and we will only be discussing the policy for the object plan store and the SQL plan store.

Determining which plans to evict is based on the cost of the plan, which will be discussed in the next section. When eviction starts is based on memory pressure. When SQL Server detects memory pressure, zero-cost

plans will be removed from cache and the cost of all other plans is reduced by half. As discussed in [Chapter 2](#) of Inside SQL Server 2005: The Storage Engine, there are two types of memory pressure that can lead to removal of plans from cache. These are referred to as local and global memory pressure.

When discussing memory pressure, we refer to the term "visible" memory. Visible memory is the directly addressable physical memory available to SQL Server buffer pool. On a 32-bit SQL Server instance, the maximum value for the visible memory is either 2 GB or 3 GB, depending on whether you have the /3 GB flag set in your boot.ini file. Memory with addresses greater than 2 GB or 3 GB is only available indirectly, through AWE-mapped-memory. On a 64-bit SQL Server instance, visible memory has no special meaning, as all the memory is directly addressable. In any of the discussion below, if we refer to visible target memory greater than 3 GB, keep in mind that is only possible on a 64-bit SQL Server. The term "target" memory refers to the maximum amount of memory that can be committed to the SQL Server process. Target memory refers to the physical memory committed to the buffer pool and is the lesser of the value you have configured for "max server memory" and the total amount of physical memory available to the operating system. So "visible target memory" is the visible portion of the target memory. Query plans can only be stored in the non-AWE-mapped memory, which is why the concept of visible memory is important. You can see a value for visible memory, specified as the number of 8-KB buffers, when you run the DBCC MEMORYSTATUS command. The section called Buffer Counts displays values for Visible memory as well as Target memory.

SQL Server defines a cachestore pressure limit value, which varies depending on the version you're running and the amount of visible target memory. We'll explain shortly how this value is used. The formula for determining the plan cache pressure limit changed in SQL Server 2005, Service Pack 2. [Table 5-3](#) shows how to determine the plan cache pressure limit in SQL Server 2000 and 2005, and indicates the change in Service Pack 2, which reduced the pressure limit with higher amounts of memory. Be aware that these formulas are subject to change again in future service packs.

Table 5-3. Determining the Plan Cache Pressure Limit

SQL Server Version	Cache Pressure Limit	SQL Server 2005 RTM & SP1	75% of visible target memory from 0-8 GB + 50% of visible target memory from 8 Gbâ 64 GB + 25% of visible target memory > 64 GB
SQL Server 2005 SP2		75% of visible target memory from 0â 4 GB + 10% of visible target memory from 4 Gbâ 64 GB + 5% of visible target memory > 64 GB	SQL Server 2000
SQL Server 2000		4 GB upper cap on the plan cache	4 GB upper cap on the plan cache

As an example, assume we are on SQL Server 2005, Service Pack 1, on a 64-bit SQL Server instance with 28 GB of target memory. The plan cache pressure limit would be 75 percent of 8 GB plus 50 percent of the target memory over 8 GB (or 50 percent of 20 GB), which is 6 GB + 10 GB or 16 GB.

On SQL Server 2005, Service Pack 2, on the 64-bit SQL Server instance with 28 GB of target memory, the plan cache pressure limit would be 75 percent of 4 GB plus 10 percent of the target memory over 4 GB (or 10 percent of 24 GB), which is 3 GB + 2.4 GB, or 5.4 GB.

Local Memory Pressure

If any single cache store grows too big, it indicates local memory pressure and SQL Server will start removing entries from that store only. This behavior prevents one store from using too much of the total system memory.

If a cache store reaches 75 percent of the cache plan pressure limit, described in [Table 5-3](#), in single-page allocations or 50 percent of the cache plan pressure limit in multipage allocations, internal memory pressure is triggered and plans will be removed from cache. For example, in the situation above we computed the cache plan pressure limit to be 5.4 GB. If any cache store exceeds 75 percent of that value, or 4.05 GB in single-page allocations, internal memory is triggered. If adding a particular plan to cache causes the cache

store to exceed the limit, the removal of other plans from cache will happen on the same thread as the one adding the new plan, which can cause the response time of the new query to be increased.

In addition to memory pressure occurring when the total amount of memory reaches a particular limit, SQL Server also indicates memory pressure when the number of plans in a store exceeds four times the hash table size for that store, regardless of the actual size of the plans. The queries below can be used to determine the number of buckets in the hash tables for the object store and the SQL store, and the number of entries in each of those stores.

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP');
SELECT type, count(*) total_entries
FROM sys.dm_os_memory_cache_entries
WHERE type IN ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
GROUP BY type;
```

Global Memory Pressure

Global memory pressure applies to memory used by all the cache stores together, and can be either external or internal. External global pressure occurs when the operating system determines that the SQL Server process needs to reduce its physical memory consumption because of competing needs from other processes on the server. All cache stores will be reduced in size when this occurs.

Internal global memory pressure can occur when virtual address space is low. Internal global memory pressure can also occur when the memory broker predicts that all cache stores combined will use more than 80 percent of the cache plan pressure limit. Again, all cache stores will have entries removed when this occurs.

As mentioned, when SQL Server detects memory pressure, all zero-cost plans will be removed from cache and the cost of all other plans is reduced by half. Any particular cycle will update the cost of at most 16 entries for every cache store. When an updated entry has a zero-cost value, it can be removed. There is no mechanism to free entries that are currently in use. However, unused dependent objects for an in-use compiled plan can be removed. Dependent objects include the executable plans and cursors, and up to half of the memory for these objects can be removed when memory pressure exists. Remember that dependent objects are inexpensive to re-create, especially compared to compiled plans.

For more information on how memory pressure is detected, please refer to [Chapter 1](#), "A Performance Troubleshooting Methodology," and [Chapter 2](#) of Inside SQL Server 2005: The Storage Engine.

Costing of Cache Entries

Evicting plans from cache is based on their cost. For adhoc plans, the cost is considered to be zero, but it is increased by one every time the plan is reused. For other types of plans, the cost is a measure of the resources required to produce the plan. When one of these plans is reused, the cost is reset to the original cost. For non-adhoc queries, the cost is measured in units called ticks, with a maximum of 31. The cost is based on three factors: I/O, context switches, and memory. Each has its own maximum within the 31-tick total.

- I/O: each I/O costs 1 tick, with a maximum of 19.
- Context switches: 1 tick each with a maximum of 8.
- Memory: 1 tick per 16 pages, with a maximum of 4.

When not under memory pressure, costs are not decreased until the total size of all plans cached reaches 50 percent of the buffer pool size. At that point, the next plan access will decrement the cost in ticks of all plans by 1. Once memory pressure is encountered, then SQL Server will start a dedicated resource monitor thread to decrement the cost of either plan objects in one particular cache (for local pressure) or all plan cache objects (for global pressure)

The DMV sys.dm_os_memory_cache_entries can show you the current and original cost of any cache entry, as well as the components that make up that cost.

```
SELECT text, objtype, refcounts, usecounts, size_in_bytes,
       disk_ios_count, context_switches_count,
       pages_allocated_count, original_cost, current_cost
  FROM sys.dm_exec_cached_plans p
  CROSS APPLY sys.dm_exec_sql_text(plan_handle)
  JOIN sys.dm_os_memory_cache_entries e
    ON p.memory_object_address = e.memory_object_address
 WHERE cacheobjtype = 'Compiled Plan'
   AND type IN ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
 ORDER BY objtype desc, usecounts DESC;
```

Note that we can find the specific entry in sys.dm_os_memory_cache_entries that corresponds to a particular plan in sys.dm_exec_cached_plans by joining on the memory_object_address column.

Objects in Plan Cache: The Big Picture

In addition to the Dynamic Management Views and Functions discussed so far, there is another metadata object called syscacheobjects that is really just a pseudotable. Prior to SQL Server 2005, there were no Dynamic Management Objects, but we did have about half a dozen of these pseudotables, including sysprocesses and syslockinfo that took no space on disk and were materialized only when someone executed a query to access them, in a similar manner to the way that Dynamic Management Objects work. These objects are still available in SQL Server 2005. In SQL Server 2000, the pseudotables are available only in the master database, or by using a full object qualification when referencing them. In SQL Server 2005, you can access syscacheobjects from any database using only the sys schema as a qualification, so we will refer to the object using its schema. [Table 5-4](#) lists some of the more useful columns in the sys.syscacheobjects object.

Table 5-4. Useful Columns in the View

Column Name	Description
bucketid	The bucket ID for this plan in an internal hash table; the bucket ID helps SQL Server locate the plan more quickly. Two rows with the same bucket ID refer to the same object (for example, the same procedure or trigger).
cacheobjtype	Type of object in the cache: Compiled Plan, Parse Tree, etc.
etc.objtype	Type of object: Adhoc, Prepared, Proc, etc.
etc.objid	One of the main keys used for looking up an object in the cache. This is the object ID stored in sysobjects for database objects (procedures, views, triggers, and so on). For cache objects such as adhoc or prepared SQL, objid is an internally generated value.
dbid	Database ID in which the cache object was compiled.
uid	The creator of the plan (for adhoc query plans and prepared plans).
refcounts	Number of other cache objects that reference this cache object.
usecounts	Number of times this cache object has been used since its creation.
pagesused	Number of memory pages consumed by the cache object.
setopts	SET option settings that affect a compiled plan. Changes to values in this column indicate that users have modified SET options.
langid	Language ID of the connection that created the cache object.
dateformat	Date format of the connection that created the cache object.
sqlModule	definition or first 3,900 characters of the batch submitted.

In SQL Server 2000, the syscacheobjects pseudotable also includes entries for executable plans. That is, the cacheobjtype column could have a value of Executable Plan. In SQL Server 2005, since executable plans are considered dependent objects and are stored completely separately from the compiled plans, they are no longer available through the sys.syscacheobjects view. To access the executable plans, you need to select directly from the sys.dm_exec_cached_plan_dependent_objects function, and pass in a plan_handle as a parameter.

As an alternative to the sys.syscacheobjects view, which is a compatibility view and is not guaranteed to exist in future versions, you can create your own view that retrieves the same information from the SQL Server 2005 Dynamic Management Objects. The script creates a view called sp_cacheobjects in the master database. Remember that any objects with a name starting with sp_, created in the master database, can be accessed from any database without having to fully qualify the object name. Besides being able to access the sp_cacheobjects view from anywhere, another benefit of creating your own object is that you can customize it. For example, it would be relatively straightforward to do one more OUTER APPLY, to join this view with the sys.dm_exec_query_plan function, to get the XML plan for each of the plans in cache.

Code View:

```
USE master
GO
CREATE VIEW sp_cacheobjects
    (bucketid, cacheobjtype, objtype, objid, dbid, dbidexec, uid,
     refcounts, usecounts, pagesused, setopts, langid, dateformat,
     status, lasttime, maxexecetime, avgexecetime, lastreads,
     lastwrites, sqlbytes, sql)
AS
SELECT pvt.bucketid,
       CONVERT(nvarchar(17), pvt.cacheobjtype) AS cacheobjtype,
       pvt.objtype,
       CONVERT(int, pvt.objectid) AS object_id,
       CONVERT(smallint, pvt.dbid) AS dbid,
       CONVERT(smallint, pvt.dbid_execute) AS execute_dbid,
       CONVERT(smallint, pvt.user_id) AS user_id,
       pvt.refcounts, pvt.usecounts,
       pvt.size_in_bytes / 8192 AS size_in_bytes,
       CONVERT(int, pvt.set_options) AS setopts,
       CONVERT(smallint, pvt.language_id) AS langid,
       CONVERT(smallint, pvt.date_format) AS date_format,
       CONVERT(int, pvt.status) AS status,
       CONVERT(bigint, 0),
       CONVERT(bigint, 0),
       CONVERT(bigint, 0),
       CONVERT(bigint, 0),
       CONVERT(bigint, 0),
       CONVERT(bigint, 0),
       CONVERT(int, LEN(CONVERT(nvarchar(max), fgs.text)) * 2),
       CONVERT(nvarchar(3900), fgs.text)
FROM (SELECT ecp.* , epa.attribute, epa.value
      FROM sys.dm_exec_cached_plans ecp
      OUTER APPLY
          sys.dm_exec_plan_attributes(ecp.plan_handle) epa) AS ecpa
PIVOT (MAX(epca.value) for epca.attribute IN
        ("set_options", "objectid", "dbid",
         "dbid_execute", "user_id", "language_id",
         "date_format", "status")) AS pvt
OUTER APPLY sys.dm_exec_sql_text(pvt.plan_handle) fgs
```

You might notice that several of the output columns are hardcoded to a value of 0. For the most part, these are columns for data that is no longer maintained in SQL Server 2005. In particular, these are columns that report on performance information for the cached plans. In SQL Server 2000, this performance data was maintained

for each batch. In SQL Server 2005, it is maintained on a statement level and available through sys.dm_exec_query_stats. To be compatible with the sys.syscacheobjects view, the new view here returns something in those column positions. If you choose to customize this view, you could choose to remove those columns.

Multiple Plans in Cache

SQL Server will try to limit the number of plans for a query or a procedure. Because plans are reentrant, this is easy to accomplish. You should be aware of some situations that will cause multiple query plans for the same procedure to be saved in cache. The most likely situation is a difference in certain SET options, as discussed previously.

One other connection issue can affect whether a plan can be reused. If an owner name must be resolved implicitly, a plan cannot be reused. For example, suppose user sue issues the following SELECT statement:

```
SELECT * FROM Orders
```

SQL Server will first try to resolve the object by looking for an object called Orders in the default schema for the user sue, and if no such object can be found, it will look for an object called Orders in the dbo schema. If user dan executes the exact same query, the object can be resolved in a completely different way (to a table in the default schema of the user dan), so sue and dan could not share the plan generated for this query. Because there is a possible ambiguity when using the unqualified object name, the query processor will not assume that an existing plan can be reused. However, the situation is different if sue issues this command:

```
SELECT * FROM dbo.Orders
```

Now there's no ambiguity. Anyone executing this exact query will always reference the same object. In the sys.syscacheobjects view, the column uid indicates the user ID for the connection in which the plan was generated. For adhoc queries, only another connection with the same user ID value can use the same plan. The one exception is if the user ID value is recorded as -2 in syscacheobjects, which indicates that the query submitted does not depend on implicit name resolution and can be shared among different users. This is the preferred method.

Tip



It is strongly recommended that objects are always qualified with their containing schema name, so that you never need to rely on implicit resolutions, and the reuse of plan cache can be more effective.

When to Use Stored Procedures and Other Caching Mechanisms

Keep the following guidelines in mind when you are deciding whether to use stored procedures or one of the other mechanisms:

- **Stored procedures** These objects should be used when multiple connections are executing batches in which the parameters are known. They are also useful when you need to have control over when a block of code is to be recompiled.

- Adhoc caching This option is beneficial only in limited scenarios. It is not dependable enough for you to design an application expecting this behavior to correctly control reuse of appropriate plans.
- Autoparameterization This option can be useful for applications that cannot be easily modified. However, it is preferable when you initially design your applications that you use methods that explicitly allow you to declare what your parameters are and what are their datatypes, such as the two suggestions below.
- The sp_executesql procedure This procedure can be useful when the same batch might be used multiple times and when the parameters are known.
- The prepare and execute method These methods are useful when multiple users are executing batches in which the parameters are known, or when a single user will definitely use the same batch multiple times.

Troubleshooting Plan Cache Issues

In order to start addressing problems with plan cache usage and management, you first must determine that existing problems are actually caused by plan caching issues. Performance problems caused by misuse or mismanagement of plan cache, or inappropriate recompilation, can manifest themselves as simply a decrease in throughput or an increase in query response time. Problems with caching can also show up as out-of-memory errors or connection time-out errors, which can be caused by all sorts of different conditions.

Wait Statistics Indicating Plan Cache Problems

In order to determine that plan caching behavior is causing problems, one of the first things to look at is your SQL Server's wait statistics. Wait statistics will be covered in more detail in [Chapter 6 "Concurrency Problems,"](#) but here we'll tell you about some of the primary wait types that can indicate problems with your plan cache.

Wait statistics are displayed when you query the sys.dm_os_wait_stats view. The query below will list all of the resources that your SQL Server service might have to wait for, and it will display the resources with the longest waiting list:

```
SELECT *
FROM sys.dm_os_wait_stats
ORDER BY waiting_tasks_count DESC
```

As discussed in [Chapter 1](#), the values shown in this view are cumulative, so if you need to see the resources being waited on during a specific time period, you will have to poll the view at the beginning and end of the period. If you see relatively large wait times for any of the following resources, or if these resources are near the top of the list returned from the query above, you should investigate your plan cache usage.

- CMEMTHREAD waits

This wait type indicates that there is contention on the memory object from which cache descriptors are allocated. A very high rate of insertion of entries into the plan cache can cause contention problems. Similarly, contention can also occur when entries are removed from cache and the resource monitor thread is blocked. There is only one thread-safe memory object from which descriptors are allocated, and as we've seen, there is only a single cache store for adhoc compiled plans.

Consider the same procedure being called dozens or hundreds of times. Remember that SQL Server 2005 will cache the adhoc shell query that includes the actual parameter for each individual call to the procedure, even though there may be only one cached plan for the procedure itself. As SQL Server

starts experiencing memory pressure, the work to insert the entry for each individual call to the procedure can begin to cause excessive waits resulting in a drop in throughput or even out-of-memory errors.

SQL Server 2005 Service Pack 2 includes some changes to caching behavior to alleviate some of the flooding of cache that can occur when the same procedure or autoparameterized query is called repeatedly with different parameters. In Service Pack 2, zero-cost batches that contain SET statements or transaction control will not be cached at all. The only exception is for those batches that contain only SET and transaction control statements. This is not that much of a loss, as plans for batches containing SET statements can never be reused in any case. Also in Service Pack 2, the memory object from which cache descriptors are allocated has been partitioned across all the CPUs to alleviate contention on the memory object which should reduce CMEMTHREAD waits.

- **SOS_RESERVEDMEMBLOCKLIST** waits

This wait type can indicate the presence of cached plans for queries with a large number of parameters, or with a large number of values specified in an IN clause. These types of queries require that SQL Server allocate in larger units, called multipage allocations. You can look at the view `sys.dm_os_memory_cache_counters` to see the amount of memory allocated in the multipage units.

```
SELECT name, type, single_pages_kb, multi_pages_kb,
       single_pages_in_use_kb, multi_pages_in_use_kb
  FROM sys.dm_os_memory_cache_counters
 WHERE type = 'CACHESTORE_SQLCP' OR type = 'CACHESTORE_OBJCP';
```

Clearing out plan cache with DBCC FREEPROCCACHE can alleviate problems caused by too many multipage allocations, at least until the queries are reexecuted and the plans are cached again. In addition, the cache management changes in SQL Server 2005 Service Pack 2 can also reduce the waits on SOS_RESERVEDMEMBLOCKLIST. You can also consider rewriting the application to use alternatives to long parameters or long IN lists. In particular, long IN lists can almost always be improved by creating a table of the values in the IN list and joining with that table.

- **RESOURCE_SEMAPHORE_QUERY_COMPILE** waits

This wait type indicates that there are a large number of concurrent compilations. In order to prevent inefficient use of query memory, SQL Server 2005 limits the number of concurrent compile operations that need extra memory. If you notice a high value for RESOURCE_SEMAPHORE_QUERY_COMPILE waits, you can examine the entries in the plan cache through the `sys.dm_exec_cached_plans` view, as shown:

```
SELECT usecounts, cacheobjtype, objtype, bucketid, text
  FROM sys.dm_exec_cached_plans
    CROSS APPLY sys.dm_exec_sql_text(plan_handle)
 WHERE cacheobjtype = 'Compiled Plan'
 ORDER BY objtype;
```

If there are no results with the objtype value of Prepared, it means that SQL Server is not autoparameterizing your queries. You can try altering the database to PARAMETERIZATION FORCED in this case, but this option will affect the entire database, including queries that might not benefit from autoparameterization. To force SQL Server to autoparameterize just certain queries, plan guides can be used. We'll discuss plan guides in the next section.

Keep in mind that caching is done on a per-batch level. If you try to force parameterization using `sp_executesql` or Prepare/Execute, all the statements in the batch must be parameterized for the plan to be reusable. If a batch has some parameterized statements and some using constants, each execution of the batch with different constants will be considered distinct, and there will be no value to the parameterization in only part of the batch.

Other Caching Issues

In addition to looking at the wait types that can indicate problems with caching, there are some other coding behaviors that can have a negative impact on plan reuse.

- Verify parameter types, both for prepared queries and autoparameterization.

With prepared queries, you actually specify the parameter datatype, so it's easier to make sure you are always using the same type. When SQL Server parameterizes, it makes its own decisions as to datatype. If you look at the parameterized form of your queries of type Prepared, you'll see the datatype that SQL Server assumed. We saw earlier in the chapter that a value of 12,345 will be assumed to be a different datatype than 12, and two queries that are identical except for these specific values will never be able to share the same autoparameterized plan.

If the parameter passed is numeric, SQL Server will determine the datatype based on the precision and scale. A value of 8.4 will have a datatype of numeric (2, 1) and 8.44 will have a datatype of numeric (3, 2). For varchar data type, server side parameterization will not be so dependent on the length of the actual value. Take a look at these two queries in the Northwind2 database.

```
SELECT * FROM Customers
WHERE CompanyName = 'Around the Horn';
GO
SELECT * FROM Customers
WHERE CompanyName = 'Rattlesnake Canyon Grocery';
GO
```

Both of these queries will be autoparameterized to the following:

```
(@0 varchar(8000)) select * from Customers where CompanyName = @0
```

- Monitor plan cache size and data cache size.

In general, as more queries are run, the amount of memory used for data page caching should increase along with the amount of memory used for plan cache. However, as we saw previously when discussing plan cache size, in SQL Server 2005 prior to Service Pack 1, the maximum limit for plan cache could grow to be up to 80 percent of the total buffer pool before memory pressure would start forcing plans to be evicted. This can result in severe performance degradation for those queries that depend on good data caching behavior. For any amount of memory greater than 4 GB, Service Pack 2 changes the size limit that plan cache can grow to before memory pressure is indicated. One of the easiest places to get a comparison of the pages used for plan cache and the pages used for data cache is the performance counters. Take a look at the following counters: SQL Server: Plan Cache\Cache Pages(_Total) and SQLServer: BufferManager\Database pages.

Troubleshooting Caching and Recompilation Summary

There are two main tools for detecting excessive compiles and recompiles. Keep in mind that compiling and recompiling are not the same thing. Recompiling is done when an existing module or statement is determined to be no longer valid or no longer optimal. All recompiles are considered compiles, but not vice versa. For example, when there is no plan in cache, or when executing a procedure using the WITH RECOMPILE option, or executing a procedure that was created WITH RECOMPILE, SQL Server considers this a compile but not a recompile. You can use either System Monitor or SQL Trace to detect compilations and recompilations.

System Monitor (Perfmon)

The SQL Statistics object allows you to monitor compilations, as well as the types of requests that are sent to your SQL Server. You can monitor the number of query compilations and recompilations along with the number of batches processed to determine if the compiles are a major factor in system resource use, especially CPU use. Ideally, the ratio of SQL Recompilations/Sec to Batch Requests/Sec should be very low. Of course, low is a relative term, but if you have baseline measurements, you can determine when this ratio increases, and use that as an indication that recompiles are increasing, and you need to do further investigation into the causes.

SQL Trace

If the System Monitor counters indicate a high number of recompiles, you would then need to look at trace data to determine which batches or stored procedures were being recompiled. The SQL Trace data can give you that information along with the reason for the recompilation. You can use the following events to get this information:

- SP:Recompile/SQL:StmtRecompile

The SP:Recompile event (in the Stored Procedures category) and the SQL:StmtRecompile event (in the TSQL category) indicate which stored procedures and statements have been recompiled. When you recompile a stored procedure, one of each of these events is generated for every statement within a stored procedure that is recompiled. Keep in mind that when a stored procedure recompiles, only the statement that caused the recompilation is recompiled. This is different from SQL Server 2000, in which the stored procedures were always recompiled in their entirety. Some of the more important data columns for the SP:Recompile event class are listed below. The EventSubClass data column is particularly useful for determining the reason for the recompile. SP:Recompile is triggered only for procedures or triggers that are recompiled, so for SQL Server 2005, it might be more useful to monitor SQL:StmtRecompile as this event class is fired when any type of batch, adhoc statement, stored procedure, or trigger is recompiled.

If these tools indicate that you have excessive compilation or recompilation, you can consider the following actions:

- If the recompile is caused by a change in a SET option, the SQL Trace text data for Transact-SQL statements immediately preceding the recompile event can indicate which SET option changed. It's best to change SET options when a connection is first made, and avoid changing them after you have started submitting statements on that connection, or inside a store procedure.
- Recompilation thresholds for temporary tables are lower than for normal tables, as we discussed earlier. If the recompiles on a temporary table are caused by statistics changes, a trace will have a data value in the EventSubclass column that indicates that statistics changed for an operation on a temporary table. You can consider changing the temporary tables to table variables, for which statistics are not maintained. Because no statistics are maintained, changes in statistics cannot induce recompilation. However, lack of statistics can result in suboptimal plans for these queries. Your own testing can determine if the benefit of table variables is worth the cost. Another alternative is to use the KEEP PLAN query hint, which sets the recompile threshold for temporary tables to be the same as for permanent tables.
- To avoid all recompilations that are caused by changes in statistics, whether on a permanent or a temporary table, you can specify the KEEPFIXED PLAN query hint. With this hint, recompilations can only happen because of correctness-related reasons, as described earlier. An example might be when a recompilation occurs if the schema of a table that is referenced by a statement changes, or if a table is marked for recompile by using the sp_recompile stored procedure.
- Another way to prevent recompiles caused by statistics changes is by turning off the automatic updates of statistics for indexes and statistics. Note, however, that turning off the autostatistics feature

is usually not a good idea. If you do, the optimizer will no longer be sensitive to data changes and is likely to come up with a suboptimal plan. This method should only be considered as a last resort after exhausting all other options.

- All Transact-SQL code should use two-part object names (for example, Inventory.ProductList) to indicate exactly what object is being referenced, which can help avoid recompilation.
- Do not use DDL within conditional constructs such as IF statements.
- Explore the Database Engine Tuning Advisor (DTA) to see if it recommends any indexing changes that might improve the execution time of your queries.
- Check to see if the stored procedure was created with the WITH RECOMPILE option. In many cases, there are only one or two statements within a stored procedure that might benefit from recompilation on every execution, and in SQL Server 2005, we can use the RECOMPILE query hint for just those statements. This is much better than using the WITH RECOMPILE option for the entire procedure, which means every statement in the procedure will be recompiled every time the procedure is executed.

Plan Guides and Optimization Hints

In Chapters 3 and 4, we looked at execution plans to determine when a query was being executed optimally. We've looked at situations in which SQL Server will reuse a plan when it might have been best to come up with a new one, and we've seen situations in which SQL Server will not reuse a plan even if there is perfectly good one in cache already. One way to encourage plan reuse that has already been discussed in this chapter is to enable the PARAMETERIZATION FORCED database option. In other situations, where we just can't get the optimizer to reuse a plan, we can use optimizer hints. Optimizer hints can also be used to force SQL Server to come up with a new plan in those cases in which it might be using an existing plan. Although there are dozens of hints that you can use in your Transact-SQL code to affect the plan that SQL Server comes up with, in this section we'll only specifically describe those hints that affect recompilation, as well as the mother of all hints, USE PLAN, which is new in SQL Server 2005. Finally, we'll discuss a new SQL Server 2005 feature called plan guides.

Optimization Hints

All of the hints that we'll be telling you about in this section are referred to in the SQL Server Books Online as Query Hints, to distinguish them from Table Hints, which are specified in your FROM clause after a table name, and Join Hints, which are specified in your JOIN clause before the word join. However, since technically all hints affect your queries, we frequently refer to query hints as Option Hints, since they are specified in a special clause called the OPTION clause, which is used just for specifying this type of hint. An OPTION clause, if included in a query, is always the last clause of any Transact-SQL statement, as you'll see in the code examples below.

Recompile

The RECOMPILE hint forces SQL Server to recompile a query. It is particularly useful when only a single statement within a batch needs to be recompiled. You know that SQL Server compiles your Transact-SQL batches as a unit, determining the execution plan for each statement in the batch, and it doesn't execute any statements until the entire batch is compiled. This means that if the batch contains a variable declaration and assignment, the assignment doesn't actually take place during the compilation phase. When the following batch is optimized, SQL Server doesn't have a specific value for the variable.

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID;
```

The plan for the SELECT statement will show that SQL Server is scanning the entire clustered index, because during optimization, SQL Server had no idea what value it was going to be searching for, and couldn't use the histogram in the index statistics to get a good estimate of the number of rows. If we had replaced the variable with the constant LAZYK, SQL Server could have determined that only a very few rows would qualify, and would have chosen to use the nonclustered index on customerID. The RECOMPILE hint can be very useful here, as it tells the SQL Server optimizer to come up with a new plan for the single SELECT statement, right before that statement is executed, which will be after the SET statement has executed.

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID
OPTION (RECOMPILE);
```

Note that a variable is not the same as a parameter, even though they are written the same way. Because a procedure is only compiled when it is being executed, SQL Server will always have a specific parameter value to use. Problems arise when the previously compiled plan is then used for different parameters. However, for a local variable, the value is never known when the statements using the variable are compiled, unless the RECOMPILE hint is used.

Optimize For

This hint tells the optimizer to optimize the query as if a particular value has been used for a variable. Execution will use the real value. Keep in mind that the OPTIMZE FOR hint does not force a query to be recompiled. It only instructs SQL Server to assume a variable or parameter has a particular value in those cases in which SQL Server has already determined that the query needs optimization. As the OPTIMIZE FOR hint was discussed in [Chapter 4](#), "Troubleshooting Query Performance," we won't say any more about it here.

Keep Plan

This hint relaxes the recompile threshold for a query, particularly for queries accessing temporary tables. As we saw earlier in this chapter, a query accessing a temp table can be recompiled when as few as six changes have been made to the table. If the query uses the KEEP PLAN hint, the recompilation threshold for temp tables is changed to be the same as for permanent tables.

Keepfixed Plan

This hint inhibits all recomiles because of optimality issues. With this hint, queries will only be recompiled when forced, or if the schema of the underlying tables is changed, as described in the section on correctness-based recomiles above.

Parameterization Simple | Forced

This hint overrides the PARAMETERIZATION option for a database. If the database is set to PARAMETERIZATION FORCED, individual queries can avoid that and only be parameterized if they meet the strict list of conditions. Alternatively, if the database is set to PARAMETERIZATION SIMPLE, individual queries can be parameterized on a case-by-case basis. Note however that the PARAMETERIZATION hint can only be used in conjunction with plan guides, which we'll discuss shortly.

Use Plan

This hint was discussed in [Chapter 4](#), as a way to force SQL Server to use a plan that you might not be able to specify using the other hints. The plan specified must be in XML format, and can be obtained from a query that uses the desired plan by using the option SET SHOWPLAN_XML ON. Because USE PLAN hints contain a complete XML document in the query hint, they are best used within plan guides, which are discussed in the next section.

Purpose of Plan Guides

Although in most cases it is recommended that you allow the SQL Server query optimizer to determine the best plan for each of your queries, there are times when the optimizer just can't come up with the best plan and you may find that the only way to get reasonable performance is to use a hint. This is usually a straightforward change to your applications, once you have verified that the desired hint is really going to make a difference. However, in some environments you have no control over the application code. In cases in which the actual SQL queries are embedded in inaccessible vendor code, or in which modifying vendor code would break your licensing agreement or invalidate your support guarantees, you might not be able to simply add a hint onto the misbehaving query.

SQL Server 2005's Plan Guides provide a solution by giving you a mechanism to add hints to a query without changing the query itself. Basically, a plan guide tells SQL Server's optimizer that if it tries to optimize a query having a particular format, it should add a specified hint to the query. SQL Server 2005 supports three kinds of plan guides: SQL, Object, and Template, which we'll explore shortly.

Plan guides are available in the Standard, Enterprise, Evaluation, and Developer Editions of SQL Server 2005. If you detach a database containing plan guides from a supported edition and attach the database to an unsupported edition, such as Workgroup or Express, SQL Server will not use any plan guides. However the metadata containing information about plan guides will still be available.

Types of Plan Guides

The three types of plan types can all be created using the sp_create_plan_guide procedure. The general form of the sp_create_plan_guide procedure is as follows:

```
sp_create_plan_guide 'plan_guide_name', 'statement_text',
    'type_of_plan_guide', 'object_name_or_batch_text',
    'parameter_list', 'hints'
```

We'll discuss each of the types of plan guides, and then we'll look at the mechanisms for working with plan guides and the metadata that keeps track of information about them.

Object Plan Guides

A plan guide of type object indicates that you are interested in a Transact-SQL statement appearing in the context of a SQL Server object, which can be a stored procedure, a user defined function, or a trigger in the database in which the plan guide is created. As an example, suppose we have a stored procedure called Sales.GetOrdersByCountry that takes a country as a parameter, and after some error checking and other validation, it returns a set of rows for the Orders placed by Customers in the specified country. Suppose further that our testing has determined that a parameter value of US gives us the best plan. Here is an example of a plan guide that tells SQL Server to use the OPTIMIZE FOR hint, whenever the specified statement is found in the Sales.GetOrdersByCountry procedure:

```

EXEC sp_create_plan_guide
    @name = N'plan_US_Country',
    @stmt =
        N'SELECT SalesOrderID, OrderDate, h.CustomerID, h.TerritoryID
         FROM Sales.SalesOrderHeader AS h
         INNER JOIN Sales.Customer AS c
             ON h.CustomerID = c.CustomerID
         INNER JOIN Sales.SalesTerritory AS t
             ON c.TerritoryID = t.TerritoryID
         WHERE t.CountryRegionCode = @Country',
    @type = N'OBJECT',
    @module_or_batch = N'Sales.GetOrdersByCountry',
    @params = NULL,
    @hints = N'OPTION (OPTIMIZE FOR (@Country = N'''US'''))';

```

Once this plan is created in the AdventureWorks database, every time the Sales.GetOrdersByCountry procedure is compiled, the statement indicated in the plan will be optimized as if the actual parameter passed was the string US. No other statements in the procedure will be affected by this plan, and if the specified query occurs outside of the Sales.GetOrdersByCountry procedure, the plan guide will not be invoked. (The companion website, which contains all the code used in all the book examples, also contains a script to build the Sales.GetOrdersByCountry procedure.)

SQL Plan Guides

A plan guide of type SQL indicates you are interested in a particular SQL statement, either as a stand-alone statement, or in a particular batch. Transact-SQL statements that are sent to SQL Server by CLR objects or extended stored procedures, or that are part of dynamic SQL invoked with the EXEC (sql_string) construct, will be processed as batches on SQL Server. To use them in a plan guide, their type should be set to SQL. For a stand-alone statement, the @module_or_batch parameter to sp_create_plan_guide should be set to NULL, so that SQL Server will assume the batch and statement have the same value. If the statement you are interested in is in a larger batch, the entire batch text needs to be specified in the @module_or_batch parameter. If a batch is specified for a SQL plan guide, the text of the batch needs to be exactly the same as it will appear in the application. The rules aren't quite as strict as those for adhoc query plan reuse, discussed earlier in this chapter, but they are close. Make sure you use the same case, the same whitespace, etc. as your application does.

Here is an example of a plan guide that tells SQL Server to use only one CPU (no parallelization) when a particular query is executed as a stand-alone query:

```

EXEC sp_create_plan_guide
    @name = N'plan_SalesOrderHeader_DOP1',
    @stmt = N'SELECT TOP 10 *
                 FROM Sales.SalesOrderHeader
                 ORDER BY OrderDate DESC',
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (MAXDOP 1)';

```

Once this plan is created in the AdventureWorks database, every time the specified statement is encountered in a batch by itself, it will have a plan created that uses only a single CPU. If the specified query occurs as part of a larger batch, the plan guide will not be invoked.

Template Plan Guides

A SQL plan guide of type Template can only use the PARAMETERIZATION FORCED or PARAMETERIZATION SIMPLE hints, to override the PARAMETERIZATION database setting. Template guides are a bit trickier to work with, as you have to have SQL Server construct a template of your query in the same format that it will be in once it is parameterized. This isn't hard, because SQL Server supplies us with a special procedure called `sp_get_query_template`, but to use template guides you need to perform several prerequisite steps. If you take a look at the two plan guide examples above, you'll see that the parameter called `@params` was NULL for both OBJECT and SQL plan guides. You only specify a value for `@params` with a TEMPLATE plan guide.

To see an example of using a template guide and forcing parameterization, first clear your procedure cache and then execute these two queries in the AdventureWorks database:

```
DBCC FREEPROCCACHE;
GO
SELECT * FROM AdventureWorks.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
```

These queries are very similar, and the plans for both are identical, but because the query is considered too complex, SQL Server will not autoparameterize them. If, after executing both queries, you look at the plan cache, you'll see only adhoc queries. If you've created the `sp_cacheobjects` view described earlier in the chapter, you could use that, otherwise, replace `sp_cacheobjects` with `sys.syscacheobjects`.

```
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';
```

To create a plan guide to force statements of this type to be parameterized, we first need to call the procedure `sp_get_query_template` and pass two variables as output parameters. One parameter will hold the parameterized version of the query, and the other will hold the parameter list and the parameter datatypes. The code below will then SELECT these two output parameters so you can see their contents. Of course, you can remove this SELECT from your own code. Finally, we call the `sp_create_plan_guide` procedure, which instructs the SQL Server optimizer to use PARAMETERIZATION FORCED anytime it sees a query that matches this specific template. In other words, anytime a query that parameterizes to the same form as the query here, it will use the same plan already cached.

```
DECLARE @sample_statement nvarchar(max);
DECLARE @paramlist nvarchar(max);
EXEC sp_get_query_template
N'SELECT * FROM AdventureWorks.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;',
@sample_statement OUTPUT,
@paramlist OUTPUT
SELECT @paramlist as parameters, @sample_statement as statement
EXEC sp_create_plan_guide @name = N'Template_Plan',
@stmt = @sample_statement,
```

```

@type = N'TEMPLATE',
@Module_or_batch = NULL,
@params = @paramlist,
@hints = N'OPTION(PARAMETERIZATION FORCED)';

```

After creating the plan guide, run the same two statements as above, and then examine the plan cache:

```

DBCC FREEPROCCACHE;
GO
SELECT * FROM AdventureWorks.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks.Sales.SalesOrderDetail AS d
    ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
GO
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';

```

You should now see a prepared plan with the following parameterized form:

```

(@0 int)select * from AdventureWorks.Sales.SalesOrderHeader as h
inner join AdventureWorks.Sales.SalesOrderDetail as d
on h.SalesOrderID = d.SalesOrderID
where h.SalesOrderID = @0

```

Managing Plan Guides

In addition to the `sp_create_plan_guide` and `sp_get_query_template` procedures, the other basic procedure for working with plan guide is `sp_control_plan_guide`. This procedure allows you to DROP, DISABLE, or ENABLE a plan guide, using the following basic syntax:

```
sp_control_plan_guide '<control_option>' [, '<' ]
```

There are six possible control_option values: DISABLE, DISABLE ALL, ENABLE, ENABLE ALL, DROP, and DROP ALL. The `plan_guide_name` parameter is optional, because with any of the ALL control_option values, no `plan_guide_name` value is supplied. Plan guides are local to a particular database, so the DISABLE ALL, ENABLE ALL, and DROP ALL values apply to all plan guides for the current database. In addition, plan guides behave like schema-bound views in a way; the stored procedures, triggers, or functions referred in any OBJECT plan guide in a database cannot be altered or dropped. So for our OBJECT plan guide above, as long as the plan guide exists, the `AdventureWorks.Sales.GetOrdersByCountry` procedure cannot be altered or dropped. This is true whether the plan guide is disabled or enabled, and will remain true until all plan guides referencing those objects are dropped with `sp_control_plan_guide`.

The metadata view that contains information about plan guides in a particular database is `sys.plan_guides`. This view contains all the information supplied in the `sp_create_plan_guide` procedure plus additional

information such as the creation date and last modification date of each plan guide. Using the information in this view, you can manually reconstruct the plan guide definition, if necessary. However, no built-in mechanism automatically scripts the plan guide definitions, as you can do with most other SQL Server objects, so it is strongly recommended that you save your actual sp_create_plan_guide scripts in case you ever need to move your definitions to a new server. Eric Hanson, from the SQL Server development team at Microsoft, has created a set of procedures to interpret the information in sys.plan_guides and generate the appropriate sp_create_plan_guide statements. These scripts are available on the companion website.

Plan Guide Considerations

In order for SQL Server to determine that there is an appropriate plan guide to use, the statement text in the plan guide must exactly match the query being compiled. This must be an exact character-for-character match, including case, white space, and comments, just like when SQL Server is determining whether it can reuse adhoc query plans, as we discussed earlier in the chapter. If your statement text is close, but not quite an exact match, this can lead to a situation that is very difficult to troubleshoot. When matching a SQL template, whether the definition also contains a batch that the statement must be part of, SQL Server does allow more leeway in the definition of the batch. In particular, keyword case, whitespace, and comments will be ignored.

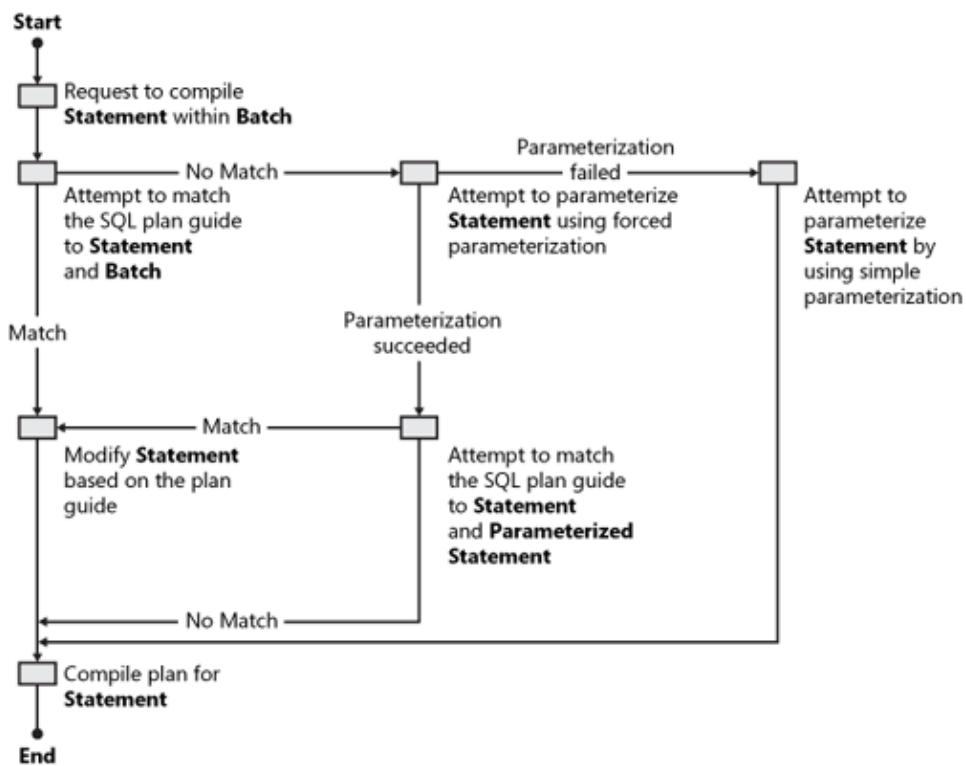
To make sure your plan guides use the exact text that is submitted by your applications, you can run a trace using SQL Server Profiler, and capture the SQL:BatchCompleted and RPC:Completed events. After the relevant batch (the one you want to create a plan guide for) shows up in the top window of your Profiler output, you can right-click the event and select Extract Event Data to save the SQL Text of the batch to a text file. It is not sufficient to copy and paste from the lower window in the Profiler, because the output there can introduce extra line breaks.

To verify that your plan guide was used, you can look at the XML plan for the query. If you can run the query directly, you can use the option SET SHOWPLAN_XML ON, or you can capture the showplan XML through a trace. An XML plan will have two specific items, indicating that the query used a plan guide. These items are PlanGuideDB and PlanGuideName. If the plan guide was a template plan guide, the XML plan will also have the items TemplatePlanGuideDB and TemplatePlanGuideName.

When a query is submitted for processing, if there are any plan guides in the database at all, SQL Server will first check to see if the statement matches a SQL plan guide or OBJECT plan guide. The query string is hashed to make it faster to find any matching strings in the database's existing plan guides. If no matching SQL or OBJECT plan guides are found, SQL Server will then check for a TEMPLATE plan guide. If it finds a TEMPLATE guide, it will then try to match the resulting parameterized query to a SQL plan guide. This gives you the possibility of applying additional hints to your queries using forced parameterization. [Figure 5-2](#), copied from SQL Server Books Online, shows the process that SQL Server will use to check for applicable plan guides.

Figure 5-2. Checking for applicable plan guides

[[View full size image](#)]



The key steps are the following, which follow the flowchart from the top left, take the top branch to the right, the middle branch down, and then right at the center, to the point where the statement is modified based on the plan guide and its hints.

1. For a specific statement within the batch, SQL Server tries to match the statement to a SQL-based plan guide, whose @module_or_batch argument matches that of the incoming batch text, including any constant literal values, and whose @stmt argument also matches the statement in the batch. If this kind of plan guide exists and the match succeeds, the statement text is modified to include the query hints specified in the plan guide. The statement is then compiled using the specified hints.
2. If a plan guide is not matched to the statement in step 1, SQL Server tries to parameterize the statement by using forced parameterization. In this step, parameterization can fail for any one of the following reasons:
 - a. The statement is already parameterized or contains local variables.
 - b. The PARAMETERIZATION SIMPLE database SET option is applied (the default setting), and there is no plan guide of type TEMPLATE that applies to the statement and specifies the PARAMETERIZATION FORCED query hint.
 - c. A plan guide of type TEMPLATE exists that applies to the statement and specifies the PARAMETERIZATION SIMPLE query hint.

Let's look at an example that involves the distribution of data in the SpecialOfferID column in the Sales.SalesOrderDetail table in the AdventureWorks database. There are 12 different SpecialOfferID values, and most of them only occur a few hundred times at most, out of the 121,317 rows in the Sales.SalesOrderDetail, as the following script and output illustrates:

```

USE AdventureWorks

SELECT SpecialOfferID, COUNT(*) as Total
FROM Sales.SalesOrderDetail
GROUP BY SpecialOfferID;
RESULTS:
SpecialOfferID      Total
-----
1                  115884
2                  3428
3                  606
4                  80
5                  2
7                  137
8                  98
9                  61
11                 84
13                 524
14                 244
16                 169

```

As there are 1,238 pages in the table, for most of the values a nonclustered index on SpecialOfferID could be useful, so here is the code to build one:

Code View:

```
CREATE INDEX Detail_SpecialOfferIndex ON Sales.SalesOrderDetail(SpecialOfferID);
```

We assume that there are very few queries that actually search for a SpecialOfferID value of 1 or 2, and 99 percent of the time the queries are looking for the less popular values. We would like the SQL Server optimizer to autoparameterize queries that access the Sales.SalesOrderDetail table, specifying one particular value for SpecialOfferID. So we will create a template plan guide to autoparameterize queries of this form:

```
SELECT * FROM Sales.SalesOrderDetail WHERE SpecialOfferID = 4;
```

However, we want to make sure that the initial parameter that determines the plan is not one of the values that might use a Clustered Index scan, namely the values one or two. So we can take the autoparameterized query produced by the sp_get_query_template procedure, and use it to first create a template plan guide, and then to create a SQL plan guide with the OPTIMIZE FOR hint. The hint will force SQL Server to assume a specific value of 4 every time the query needs to be reoptimized.

Code View:

```

USE AdventureWorks;
-- Get plan template and create plan Guide
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
    N'SELECT * FROM Sales.SalesOrderDetail WHERE SpecialOfferID = 4',
    @stmt OUTPUT,
    @params OUTPUT
--SELECT @stmt as statement -- show the value when debugging
--SELECT @params as parameters -- show the value when debugging

EXEC sp_create_plan_guide N'Template_Plan_for_SpecialOfferID',

```

```

@stmt,
N'TEMPLATE',
NULL,
@params,
N'OPTION (PARAMETERIZATION FORCED)';

EXEC sp_create_plan_guide
    @name = N'Force_Value_for_Prepared_Plan',
    @stmt = @stmt,
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = @params,
    @hints = N'OPTION (OPTIMIZE FOR (@0 = 4))';
GO

```

You can verify that the plan is being autoparameterized, and optimized for a value that uses a nonclustered index on SpecialOfferID by running a few tests:

```

DBCC FREEPROCCACHE;
SET STATISTICS IO ON;
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 3;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 4;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 5;
GO

```

You should note in the STATISTICS IO output that each execution uses a different number of reads, because it is finding a different number of rows through the nonclustered index. You can also verify that SQL Server is using the prepared plan by examining the STATISTICS XML output. If you set that option on, and run the query looking for a value of 5, you should have a node in your XML document very much like this:

```

<ParameterList>
<ColumnReference Column="@0"      ParameterCompiledValue="(4)"
                  ParameterRuntimeValue="(5)" />
</ParameterList>

```

Plan guides are not intended to speed up query compilation time. Not only does SQL Server first have to determine if there is a plan guide that could be a potential match for the query being compiled, but the plan enforced by the plan guide has to be one that the optimizer would have come up with on its own. In order to know that the forced plan is valid, the optimizer has to go through most of the process of optimization. The benefit of plan guides is to reduce execution time for those queries in which the SQL Server optimizer is not coming up with the best plan on its own.

Summary

For all of the caching mechanisms, reusing a cached plan avoids recompilation and optimization. This saves compilation time, but it means that the same plan is used regardless of the particular parameter values passed

in. If the optimal plan for a given parameter value is not the same as the cached plan, the optimal execution time will not be achieved. For this reason, SQL Server is very conservative about autoparameterization. When an application uses sp_executesql, prepare and execute, or stored procedures, the application developer is responsible for determining what should be parameterized. You should parameterize only constants whose range of values does not drastically affect the optimization choices.

In this chapter, we looked at the caching and reuse of plans generated by the optimizer. SQL Server can cache and reuse plans not only from stored procedures, but also from adhoc and autoparameterized queries. Because generating query plans can be expensive, it helps to understand how and why query plans are reused and when they must be regenerated. Understanding how caching and reusing plans work will help you determine when using the cached plan can be a good thing, and when you might need to make sure SQL Server comes up with a new plan to give your queries and applications the best performance.

Chapter 6. Concurrency Problems

â Ron Talmage

In this chapter:	
New Tools for Troubleshooting Concurrency	334
Troubleshooting Locking	336
Troubleshooting Blocking	340
Troubleshooting Deadlocking	354
Troubleshooting Row-Versioningâ Based Snapshot-Based Isolation Levels	369
Summary	393

You can tune your application's database queries and optimize your database server, and still not have done enough to ensure good database performance in production. This is because problems can arise from concurrently (that is, simultaneously) executing queries on the system. In fact, no amount of single-query tuning can anticipate all the problems that might occur when queries are run simultaneously on a production database system. You can have the best-tuned, fastest-running queries imaginable, but if they can't get the data they need because of concurrency problems your tuning efforts will not be enough.

Essentially, you also have to make sure that your database queries, and the connections that execute them, must be able to execute concurrently without errors (such as deadlocks or update conflicts) and without unacceptable wait times. When they don't, you will need to address concurrency issues to find out why queries that when run singly behave correctly, instead exhibit problems when they are run together.

Troubleshooting concurrency is an empirical process. It is not easy to simulate production conditions, and not easy to predict how queries will behave when they are run concurrently, that is, from many simultaneous connections. That's why you often have to take the system as it is and then diagnose concurrency problems as they occur.

For troubleshooting concurrency issues, in most cases a SQL Server process (or task) is waiting for a resource of some kind that another process has yet to release. If both processes were running serially, we would not see any contention; it's when the processes run concurrently that we see contention for some kind of resource. These waits are most often caused by requests for locks on user resources.

In this chapter, you'll learn about techniques for troubleshooting concurrency issues. In general terms, this means diagnosing and troubleshooting locking, blocking, and deadlocking situations. The basic methodology is straightforward:

- Identify the concurrency problem.
- Analyze and discover the cause.
- Make changes to resolve the issue.

Taking this approach requires that you are already familiar with considerable background details. This chapter assumes you've read or are otherwise already familiar with the content of Chapter 8, "Locking and Concurrency," in Inside SQL Server 2005: The Storage Engine, by Kalen Delaney (see the reference to this work in the bibliography, "[Additional Resources and References](#)"). Specifically that chapter covers required details about SQL Server 2005 transactions, isolation levels, and locking that are needed for troubleshooting concurrency issues.

New Tools for Troubleshooting Concurrency

SQL Server 2005 adds some important new tools for troubleshooting concurrency issues. It still includes legacy tools such as system stored procedures like sp_who2, Perfmon counters, and the SQL Trace/Profiler tool for troubleshooting concurrency problems, but expands your toolkit even further.

For detection and analysis of concurrency problems, SQL Server 2005's new tools include:

- DMVs, in particular sys.dm_os_wait_stats and sys.dm_os_waiting_tasks.
- The enhanced SQLDiag.exe utility, which gathers information about a running system.
- Row-versioning-based isolation options (the SNAPSHOT and READ COMMITTED SNAPSHOT isolation levels)
- The new 1222 trace flag and the Deadlock Graph event class in SQL Trace for detecting and analyzing deadlocks.
- The Blocked Process Report Event Class in SQL Trace/Profiler, and the blocked process threshold server configuration option for detecting long blocking.
- New counters in the SQLServer: Transactions Perfmon object that detect update conflicts in Snapshot Isolation transactions, and measure version store usage in tempdb.

For now, it's worth calling attention to the two most fundamental of these new tools: the sys.dm_os_waiting_tasks DMV for detection, and row-versioning-based isolation levels for resolution. We'll just summarize them here, and then we'll make much more use of details about them when looking at specific troubleshooting techniques.

New Blocking Detection Tool:

The most important new SQL Server 2005 tool we have for detecting blocking types of concurrency issues is sys.dm_os_waiting_tasks. This DMV shows the waiting information for all tasks. Not all waiting tasks are blocking, because tasks may wait on I/O or memory grants. When one task is waiting on another task for a relatively long time, it is being blocked. When a task is blocked by another task, the blocking task will also be listed in the view.

One advantage of this view is that it lists tasks and processes. The task represents a more granular detail of SQL Server execution than the spid (server process id). A given spid may consist of several simultaneous tasks, if the process has been parallelized; if the spid has not been parallelized, then there will be one task per spid.

Tasks can be waiting for any number of reasons, and many of those possible reasons have nothing to do with concurrency. So it's important to filter out irrelevant rows from the view to better focus on blocking issues caused by locked resources.

New Blocking Resolution Tool: Row-Versioningâ Based Isolation Levels

The other major new tool consists of using the row-versioningâ based Snapshot Isolation options, which reduce the amount of locking and therefore blocking and deadlocking. Specifically, row versioning can reduce or remove the shared locks taken by queries within and outside of transactions. As a result, queries and transactions that might have been blocked because they would issue shared locks will no longer be blocked. Unlike using the NOLOCK hint, those queries will only read committed data.

Using row-versioningâ based isolation levels has some side effects, which include increased activity in tempdb for row versioning, coding changes that may be required for triggers when using either or both of the Snapshot Isolation options, and conflicts with potential lost updates and DDL operations when using the SNAPSHOT isolation level.

Types of Concurrency Issues

When one SQL Server process or task is waiting, often it is waiting for some kind of resource that is locked by another SQL Server process. The fundamental types of concurrency issues you are most likely to confront are related to locking and include the following:

- Locking
- Blocking
- Deadlocking
- tempdb usage
- Update conflicts

The first three of these are related: locking is the basic concurrency mechanism in SQL Server, blocking occurs when one task must wait for a resource locked by another, and deadlocks occur when you have tasks mutually blocking each other.

The latter two in the list arise from using row versioning. You may see increased tempdb activity and perhaps some performance issues related to the use of row versioning in general. (You may also need to address other concurrency issues related to tempdb, such as high tempdb activity related to a high frequency of temporary table creation.) In addition, you may find cases of update conflicts when you use the SNAPSHOT isolation level for transactions that write to the database.

Chapter 6. Concurrency Problems

â Ron Talmage

In this chapter:	
New Tools for Troubleshooting Concurrency	334
Troubleshooting Locking	336
Troubleshooting Blocking	340
Troubleshooting Deadlocking	354

Troubleshooting Row-Versioning-Based Snapshot-Based Isolation Levels	369
Summary	393

You can tune your application's database queries and optimize your database server, and still not have done enough to ensure good database performance in production. This is because problems can arise from concurrently (that is, simultaneously) executing queries on the system. In fact, no amount of single-query tuning can anticipate all the problems that might occur when queries are run simultaneously on a production database system. You can have the best-tuned, fastest-running queries imaginable, but if they can't get the data they need because of concurrency problems your tuning efforts will not be enough.

Essentially, you also have to make sure that your database queries, and the connections that execute them, must be able to execute concurrently without errors (such as deadlocks or update conflicts) and without unacceptable wait times. When they don't, you will need to address concurrency issues to find out why queries that when run singly behave correctly, instead exhibit problems when they are run together.

Troubleshooting concurrency is an empirical process. It is not easy to simulate production conditions, and not easy to predict how queries will behave when they are run concurrently, that is, from many simultaneous connections. That's why you often have to take the system as it is and then diagnose concurrency problems as they occur.

For troubleshooting concurrency issues, in most cases a SQL Server process (or task) is waiting for a resource of some kind that another process has yet to release. If both processes were running serially, we would not see any contention; it's when the processes run concurrently that we see contention for some kind of resource. These waits are most often caused by requests for locks on user resources.

In this chapter, you'll learn about techniques for troubleshooting concurrency issues. In general terms, this means diagnosing and troubleshooting locking, blocking, and deadlocking situations. The basic methodology is straightforward:

- Identify the concurrency problem.
- Analyze and discover the cause.
- Make changes to resolve the issue.

Taking this approach requires that you are already familiar with considerable background details. This chapter assumes you've read or are otherwise already familiar with the content of Chapter 8, "Locking and Concurrency," in Inside SQL Server 2005: The Storage Engine, by Kalen Delaney (see the reference to this work in the bibliography, "[Additional Resources and References](#)"). Specifically that chapter covers required details about SQL Server 2005 transactions, isolation levels, and locking that are needed for troubleshooting concurrency issues.

New Tools for Troubleshooting Concurrency

SQL Server 2005 adds some important new tools for troubleshooting concurrency issues. It still includes legacy tools such as system stored procedures like sp_who2, Perfmon counters, and the SQL Trace/Profiler tool for troubleshooting concurrency problems, but expands your toolkit even further.

For detection and analysis of concurrency problems, SQL Server 2005's new tools include:

- DMVs, in particular sys.dm_os_wait_stats and sys.dm_os_waiting_tasks.
- The enhanced SQLDiag.exe utility, which gathers information about a running system.

- Row-versioning-based isolation options (the SNAPSHOT and READ COMMITTED SNAPSHOT isolation levels)
- The new 1222 trace flag and the Deadlock Graph event class in SQL Trace for detecting and analyzing deadlocks.
- The Blocked Process Report Event Class in SQL Trace/Profiler, and the blocked process threshold server configuration option for detecting long blocking.
- New counters in the SQLServer: Transactions Perfmon object that detect update conflicts in Snapshot Isolation transactions, and measure version store usage in tempdb.

For now, it's worth calling attention to the two most fundamental of these new tools: the sys.dm_os_waiting_tasks DMV for detection, and row-versioning-based isolation levels for resolution. We'll just summarize them here, and then we'll make much more use of details about them when looking at specific troubleshooting techniques.

New Blocking Detection Tool:

The most important new SQL Server 2005 tool we have for detecting blocking types of concurrency issues is sys.dm_os_waiting_tasks. This DMV shows the waiting information for all tasks. Not all waiting tasks are blocking, because tasks may wait on I/O or memory grants. When one task is waiting on another task for a relatively long time, it is being blocked. When a task is blocked by another task, the blocking task will also be listed in the view.

One advantage of this view is that it lists tasks and processes. The task represents a more granular detail of SQL Server execution than the spid (server process id). A given spid may consist of several simultaneous tasks, if the process has been parallelized; if the spid has not been parallelized, then there will be one task per spid.

Tasks can be waiting for any number of reasons, and many of those possible reasons have nothing to do with concurrency. So it's important to filter out irrelevant rows from the view to better focus on blocking issues caused by locked resources.

New Blocking Resolution Tool: Row-Versioning-Based Isolation Levels

The other major new tool consists of using the row-versioning-based Snapshot Isolation options, which reduce the amount of locking and therefore blocking and deadlocking. Specifically, row versioning can reduce or remove the shared locks taken by queries within and outside of transactions. As a result, queries and transactions that might have been blocked because they would issue shared locks will no longer be blocked. Unlike using the NOLOCK hint, those queries will only read committed data.

Using row-versioning-based isolation levels has some side effects, which include increased activity in tempdb for row versioning, coding changes that may be required for triggers when using either or both of the Snapshot Isolation options, and conflicts with potential lost updates and DDL operations when using the SNAPSHOT isolation level.

Types of Concurrency Issues

When one SQL Server process or task is waiting, often it is waiting for some kind of resource that is locked by another SQL Server process. The fundamental types of concurrency issues you are most likely to confront are related to locking and include the following:

- Locking

- Blocking
- Deadlocking
- tempdb usage
- Update conflicts

The first three of these are related: locking is the basic concurrency mechanism in SQL Server, blocking occurs when one task must wait for a resource locked by another, and deadlocks occur when you have tasks mutually blocking each other.

The latter two in the list arise from using row versioning. You may see increased tempdb activity and perhaps some performance issues related to the use of row versioning in general. (You may also need to address other concurrency issues related to tempdb, such as high tempdb activity related to a high frequency of temporary table creation.) In addition, you may find cases of update conflicts when you use the SNAPSHOT isolation level for transactions that write to the database.

Troubleshooting Locking

SQL Server locks resources to ensure the logical consistency of the database. Locking in SQL Server is not something that physically affects a data resource such as a row, page, table, or index: It is more like a reservation system that all tasks respect when they want to reserve some resource within the database. Excessive numbers of locks or locks of very long duration can lead to blocking and other problems, but locking by itself may present some issues.

Troubleshooting Lock Memory

You can determine the amount of memory that SQL Server is using for locks by monitoring the Lock Memory (KB) counter in the SQL Server:Memory Manager object in System Monitor (Perfmon). You can change the amount of memory that SQL Server allocates to locks by setting the locks option in sp_configure. You can get more granular detail about locking behavior using the SQLServer:Locks counters.

If you run out of memory for locks on your system, and SQL Server cannot allocate more memory for locks, your session will receive message 1204:

Code View:

```
The instance of the SQL Server Database Engine cannot obtain a LOCK resource at this time.
Rerun your statement when there are fewer active users. Ask the database administrator to
check the lock and memory configuration for this instance, or to check for long-running
transactions.
```

This message is fairly self-explanatory. You may need to increase the amount of memory allocated for locks, or reduce the amount of locking on the system.

If you see high memory usage for locks, you should attempt to find the underlying causes of so many locks first. It may be that SQL Server is not escalating locks enough, for example. As a rule, you should not modify the locks configuration. Once you change the locks dynamic configuration, you affect how lock escalation behaves, and therefore you may cause unintended side effects.

If you have a database that does not require any write access, consider making it read-only. That may help reduce the number of locks being taken in the system. In a read-only database, SQL Server will still issue a

shared lock on the database, and an intent shared lock on tables that it reads, but row and page locks, as well as range locks for the SERIALIZABLE isolation level, will not be issued. For example, a database on a reporting server that is only updated at night could be made read-only during the day when users are querying it. The impact of their queries on lock memory, as well as the work SQL Server's lock manager must do, would be reduced. You can also create a database snapshot of a read-only database, on the same server, and SQL Server will not issue shared locks on the database snapshot.

To reduce lock memory, also consider separating reads from writes. One way is to separate reporting from an OLTP system by creating a reporting server and using transactional replication or SQL Server Integration Services (SSIS) to get the data to another server that users can query for read purposes. That will remove the shared locking from the main OLTP server. If the database server can support it, you could consider a database snapshot to offload the reads on a periodic basis. Also, as you will see later in this chapter, you may also use one of the row-versioning-based Snapshot Isolation levels to reduce the amount of locking taken by queries reading the data.

Lock Timeout

By default, a blocked query will wait indefinitely for a lock request to be satisfied. You can instruct a session's locks to wait only up to a specified number of seconds using the LOCK_TIMEOUT setting. When the lock times out, your session will receive message 1222:

```
Lock request time out period exceeded.
```

Using LOCK_TIMEOUT is problematic for transactions, because when error 1222 occurs, SQL Server only cancels the current statement and does not abort the transaction. Therefore you should use TRY/CATCH blocks in your Transact-SQL code and trap for error 1222. You may need to roll back the transaction if the timeout occurs. For more information, see "Setting a Lock Timeout" in Inside SQL Server: The Storage Engine, Chapter 8.

Lock Escalation

Often SQL Server will lock individual rows in a table, and this is especially true if your updates and deletes affect a smaller number of rows. But when you perform mass updates, SQL Server may choose to escalate locks on a row or page to a table, in order to achieve a more optimal use of lock memory resources. Sometimes, however, you may experience blocking caused by lock escalation and may want to reduce the amount of lock escalation (see the KB article 323630, "How to resolve blocking problems that are caused by lock escalation in SQL Server" for more details).

There are a couple of ways to detect lock escalation. The easiest way is to use the Lock:Escalation event class in SQL Trace/Profiler. When an escalation occurs, the event will fire. However, you may see multiple firings for a single escalation, so it's important to tie them together.

Be sure to choose the default columns for the Lock:Escalation event class, which give you basic information. But you may also find it useful to add the columns for the TransactionID, DatabaseID, DatabaseName, and ObjectID. Because you may see multiple rows in your trace for a single escalation event, you can tie them together using TransactionID, and to a particular object (that is, a table) using ObjectID.

You can detect that escalation might be occurring by monitoring the number of table locks and their duration. If you can expect that your application would seldom, if ever, require an actual shared or exclusive lock on a table, then you could infer that whenever you do see such a lock, it is likely caused by lock escalation. You can detect table locks at a given point in time using the sys.dm_tran_locks DMV. The following query shows

an example:

```

SELECT
    request_session_id,
    resource_type,
    DB_NAME(resource_database_id) AS DatabaseName,
    OBJECT_NAME(resource_associated_entity_id) AS TableName,
    request_mode,
    request_type,
    request_status
FROM sys.dm_tran_locks AS TL
JOIN sys.all_objects AS AO
    ON TL.resource_associated_entity_id = AO.object_id
WHERE request_type = 'LOCK'
    AND request_status = 'GRANT'
    AND request_mode IN ('X', 'S')
    AND AO.type = 'U'
    AND resource_type = 'OBJECT'
    AND TL.resource_database_id = DB_ID();

```

The above query, used to find table locks, references the sys.all_objects catalog view, so the information returned is scoped to the database that the query is run under. Because sys.dm_tran_locks does not return more granular information about the object locked, there is no way to tell from it whether the object is a table. Consequently, you have to join with something in the database that will return that information, and in this case sys.all_objects contains it, and the OBJECT_NAME() function can return the name of the table. (See [Chapter 1](#), "A Performance Troubleshooting Methodology," for an example.) However, these both will only return information from the current database. Therefore, the last condition of the query filter restricts the returned rows to those resources in the current database.

Another strategy would be to use the sp_lock system stored procedure, which will return the lock type so that you can just look at table locks. Unfortunately, in order to filter sp_lock, you have to capture the data temporarily and then query it and filter in a WHERE clause. You could extract the key parts of the sp_lock stored procedure and run it yourself, but it is preferable just to query the sys.dm_tran_locks DMV and apply your filter to it.

Resolving Lock Escalation

The easiest way to prevent unwanted lock escalation is to reduce the batch sizes of mass inserts, updates, or deletes. If you must do a mass update, for example, you can limit it to a certain number of rows, or a maximum of 5,000 locks. You'll need to test this to try to find the right number that will prevent escalation. SQL Server's query optimizer can detect that you will be iterating through the table and escalate the locks anyway.

By far the most interesting method of preventing lock escalation per table is to create an intent lock on a table so that SQL Server cannot escalate the lock. Often there may be only a few or just one query that suffers from lock escalation, and you can zero in on the table in question. The Microsoft KB article 323630, "How to resolve blocking problems that are caused by lock escalation in SQL Server" gives a good example. Suppose you want to prevent lock escalation on the Sales.SalesOrderDetail table in the SQL Server 2005 sample AdventureWorks database. For example, the following code will prevent lock escalation on the table for one hour.

```

BEGIN TRAN
SELECT *
FROM Sales.SalesOrderDetail WITH (UPDLOCK, HOLDLOCK)
WHERE 1=0;
WAITFOR DELAY '1:00:00'

```

```
COMMIT
```

This query will prevent lock escalation on the Sales.SalesOrderDetail table (though it will not cause the transaction log to grow any more than without it). You will still see Lock:Escalation events in SQL Profiler when the escalations are attempted, but by inspecting sys.dm_tran_locks, you can verify that only row locks are taken by the transaction. Unfortunately, this requires keeping a transaction open indefinitely on the table, even though it does not lock any row. Also, if that table has triggers or foreign keys to other tables, SQL Server may still escalate locks on them, so stopping lock escalation on a single table may not be as simple as you hoped.

A risky option is to just turn off lock escalation entirely. You can, for example, set trace flag 1211, which disables lock escalation for the entire SQL Server instance. The problem is that while this option may reduce blocking, it will cause more locks to be taken, and, therefore, will likely increase the amount of lock memory required. If your system runs out of lock memory, it can cause SQL Server to stall or at least degrade its performance. You can also use trace flag 1224, which will disable lock escalation until the lock manager uses up to 40 percent of the non-AWE dynamically allocated memory of the SQL Server instance. You will then run out of memory for locks if the amount of lock memory reaches 60 percent of available non-AWE memory.

Another method for reducing lock escalation is to use the ROWLOCK or PAGLOCK locking hints on queries. This must be done per query and per table. The problem you may face here is the same as for all locking hints: You may be preventing your query from using a more optimal plan. If you specify a PAGLOCK hint, SQL Server may still escalate its locking to the table lock level, and you close off the use of row locks in other situations where escalation is not required and row locks would produce better behavior.

You can also set index options using the CREATE/ALTER INDEX statement in SQL Server 2005, which replaces some of the behavior of the sp_indexoption system stored procedure. Essentially you can prevent row or page locks on an index, using the SET option and setting ALLOW_ROW_LOCKS or ALLOW_PAGE_LOCKS to OFF. The default for both is ON. These options effectively allow you to control the granularity of locking of index leaf nodes, forcing SQL Server to lock at a higher granularity to begin with. Again, the value of these options in general is not high, because they prevent query plans that might benefit from using a lower granularity of locking.

If the lock escalation is occurring because of reads, you can try to remove read activity from the database by using a database snapshot, replicating to a reporting server, or using one of the Snapshot Isolation levels.

Troubleshooting Blocking

Blocking is a general term that can have several meanings, but in the context of concurrency, it refers to SQL Server processes that are prevented from completing their tasks because they are waiting on activities from other processes. More specifically, blocking usually refers to user processes that are caused to wait by other user processes because they need to access and lock data resources such as rows, pages, indexes, or tables in a database.

A task may also wait for latches, which are lower-level synchronization primitives. Such latch waits usually relate to I/O or other server configuration issues, and not to the type of blocking we are focusing on here. For more information on waiting caused by latches, see "Waiting and Blocking Issues," by Santeri Voutilainen (see the reference to SQL Server 2005 Practical Troubleshooting: The Database Engine in the bibliography, "[Additional Resources and References](#)"). Lock-based blocking is much more common and will be the main focus of this section.

Troubleshooting blocking differs from troubleshooting locking, in that you are not so much concerned about the amount of lock memory or the behavior of locking on the system, but more concerned about tasks that are waiting on other tasks because of long-held locks. Blocking is a type of waiting, where one task or process is waiting on another to complete. Blocking is just one type of waiting: there are many types of waits in SQL Server that may impact performance but are not considered blocking, such as waits for I/O and for memory. We refer to a task or process being blocked by another when it actually attempts to lock a resource that another process already has locked, and the lock types are incompatible. There are many types of locks, and several granularity levels, and numerous combinations are not compatible. When two tasks attempt to acquire incompatible locks on the same data resource (a row, page, table, or index), one task will be granted its request and the other task must wait. When the first task releases its lock, the second task can acquire its lock.

It's useful to separate the types of lock-based blocking into writer/writer blocking versus reader/writer blocking. We commonly refer to the SQL statements involved as either readers (requesting or holding shared locks for the purpose of reading data) or writers (requesting or holding an exclusive lock for purposes of inserting, updating, or deleting data).

In general, shared locks on the same resource are compatible with each other so readers will not block other readers. Exclusive locks are not compatible with other exclusive locks or with shared locks on the same resource, so writers can block readers and readers can block writers. Making things more complex, writers sometimes need to read during their activity (such as when a foreign key constraint must be checked, or an index must be read in order to find a row to change) and that can also lead to contention over locked resources.

All changes to data in SQL Server are transaction-based. Every change to data via a single INSERT, UPDATE, or DELETE statement in a batch is treated as an autocommit transaction. You can also explicitly define a transaction with your own BEGIN TRANSACTION statement. In all SQL Server transactions, exclusive locks will be held until the transaction completes helping to preserve the ACID (atomicity, consistency, isolation, and durability) properties of the transaction. (See [Chapter 1](#), "A Performance Troubleshooting Methodology," as well as *Inside SQL Server 2005: The Storage Engine*, Chapter 8 for more details.) Therefore the longer transactions take in a database, the longer exclusive locks will be held, and the greater the chance that writers will block readers. In addition, to ensure that SELECT statements and implicit lookups by other DML statements read only committed data, SQL Server uses shared locks for readers. Shared locks held on a resource are incompatible with exclusive locks, so long-running SELECTs that hold their locks can cause readers to block writers. In transactions, shared locks are normally (in the READ COMMITTED default isolation level) released immediately, whereas exclusive locks are held from the time they are granted until the transaction ends, either through a COMMIT or ROLLBACK.

Very often blocking involves more than two processes. This can happen in two ways. Multiple processes may be waiting on a single resource, such as a number of readers waiting for a writer to release its exclusive lock. In this case, one process is blocking several others. Another scenario occurs when one process blocks a second, and the second blocks a third, and so on, producing a blocking chain. In this case, the root problem is the blocking done by the first process.

Brief and momentary blocking may be normal in an active SQL Server database, though the amount will vary depending upon the type of workload. When tasks are blocked for longer periods of time, or become noticeable because queries do not perform as well as expected, your database may not be responding to its requests as quickly as it should. Lock-based blocking may be problematic in that the contention causes the overall throughput of the system to slow because the blockers are preventing other tasks from completing in a timely manner.

As a result, in troubleshooting blocking, it's important to isolate just the blocking that is long-running and problematic, and ignore brief and normal blocking.

Detecting Blocking Problems

SQL Server 2005 gives you a wealth of tools for detecting and diagnosing blocking. In addition to new tools like specialized DMVs, there is the new Blocked Process Report in SQL Trace, and the revised SQLDiag utility. All of them can be used to detect lock-based blocking. Let's start at a high level, looking at ways to quickly detect blocking, and then proceed to isolating the causes and then providing resolutions.

Quickly Detecting Blocking with System Monitor Counters

You can use System Monitor (Perfmon) to determine at a glance whether there is blocking caused by locks on your SQL Server 2005 instance. The Processes blocked counter in the SQLServer:General Statistics object will show the number of blocked processes. You can then add counters such as the Lock Waits counter from the SQLServer:Wait Statistics object to determine the count and duration of the type of waiting that is occurring. The System Monitor counters provide summary numbers only, so to drill down further you have to take a closer look into the system. Nevertheless, the Processes blocked counter gives you an idea of the size of the problem and how that blocking behaves throughout a period of time.

You can gain accurate and detailed information about blocking from DMVs such as sys.dm_os_waiting_tasks and sys.dm_tran_locks. Nevertheless, you can still use the sp_who and sp_who2 system stored procedures to monitor processes and focus on the blk or BlkBy columns to find blocker and blocking processes. However, the sp_who results only report the sessions involved and not what the queries are or what type of locks may be causing the problem. You can further inspect the sysprocesses system view for data such as whether the process is in a transaction, and you can still use the legacy DBCC INPUTBUFFER command to inspect the batch of commands currently being executed by the session, though the sys.dm_exec_requests DMV is more versatile because you can filter it yourself. However, session-level detail is limited because it does not help distinguish the multiple tasks that a parallelized query might be executing.

Note



You may occasionally see a session or spid apparently blocking itself when you use the sp_who2 stored procedure. The spid is not really blocking itself based on a locked resource. Generally this occurs with parallelized queries and simply means that the spid is waiting for I/O.

Using the DMV

The sys.dm_os_waiting_tasks DMV returns a formatted list of all waiting tasks, along with the blocking task if it is known. [Table 6-1](#) summarizes the columns returned from sys.dm_os_waiting_tasks. It's adapted from SQL Server 2005 Books Online with some additional comments added to help clarify the meaning of the columns.

Table 6-1. The Output Columns from

Column	Data type	Comment
waiting_task_address	varbinary(8)	The waiting task's memory address, which allows you to distinguish multiple tasks within a session
session_id	smallint	Also known as the session's spid. Use it to join with sys.dm_exec_requests
exec_context_id	int	Execution context id of the waiting task: 0 is the main or parent thread
wait_duration_ms	int	The wait duration in milliseconds
wait_type	varchar(60)	The wait type of the current waiting task
resource_address	varbinary(8)	Memory address of the resource the task is waiting for. Use it to join with sys.dm_tran_locks on
lock_owner_address	varbinary(8)	blocking_task_address
blocking_task_address	varbinary(8)	The blocking task's memory address, if

availableblocking_session_idsmallintBlocker's session id if available. Negative integers â 2, â 3, â 4 have special meaning and are explained below.blocking_exec_context_idintExecution context id of the blocking taskresource_descriptionnvarchar(1024)Textual description of the resource the task is waiting on

There are a number of distinct advantages to using the sys.dm_os_waiting_tasks DMV (or a filtered view of that DMV) over the sp_who or sp_who2 stored procedures, or the sysprocesses view for detecting blocking problems.

- Most important is that on a busy system you do not have to wade through tasks or sessions that are not involved in blocking. The view only shows those tasks that are waiting.
- Also sys.dm_os_waiting_tasks returns information at the task level, which is more granular than the session level. If a query has been parallelized, and one of its tasks is blocking or being blocked, sys.dm_os_waiting_tasks will reveal which task is actually involved in the blocking, in addition to the session.
- Information about the blocker is also shown.
- Another great advantage is that sys.dm_os_waiting_tasks returns the duration of the wait, so that you can filter on the view in order to just look at waits that are long enough to be of concern.

There are some conditions where the blocking_session_id may not refer to an existing session id or spid. As mentioned in the SQL Server 2005 Books Online discussion of sys.dm_os_waiting_tasks, sometimes the column may be NULL because there is no blocking session, or it cannot be identified. When you are inspecting lock blocking on a multiuser system, this should not be very common. However, there are some conditions in which SQL Server will report the blocking_session_id as a negative number. There are three possible codes for when the session id might be negative:

- â 2 = The locked resource is owned by an orphaned distributed transaction.
- â 3 = The locked resource is owned by a deferred recovery transaction.
- â 4 = For a latch wait, internal latch state transitions prevent identification of the session id

Therefore you need to be alert to whether a session id value actually appears in the blocking_session_id column.

By setting a threshold duration you can filter out waits that you are not interested in, and focus on what is likely to be long blocking. For example, the following query shows just those waits that have been occurring for more than 5 seconds:

```
SELECT
    WT.session_id AS waiting_session_id,
    WT.waiting_task_address,
    WT.wait_duration_ms,
    WT.wait_type,
    WT.blocking_session_id,
    WT.resource_description
FROM sys.dm_os_waiting_tasks AS WT
WHERE WT.wait_duration_ms > 5000;
```

When you view the results, you'll notice that the resource_description column contains a concatenated set of strings with information about the blocking session. For example, with an example block on a row in the HumanResources.Department table of the AdventureWorks database, the resource_description column contains, all in one string:

- keylock
- hobtid=72057594044022784
- dbid=8

- id=lock80130180
- mode=X
- associatedObjectId=72057594044022784

This tells you that the type of lock is a key lock, the database id is 8, and the blocking session has an exclusive lock granted.

The sys.dm_os_waiting_tasks DMV will report all waiting tasks, whether they are related to locks and blocking. Some of the waiting may be unrelated to blocking, and have more to do with I/O or memory contention. To refine your focus to just lock-based blocking, you can query the sys.dm_tran_locks DMV.

Using the DMV

When sessions block each other, it's because one session has a resource locked that another session needs. This lock-based blocking is viewable using the sys.dm_tran_locks DMV. Because it returns information about all locks, the number of rows may be large; in addition, a lot of columns are returned and some are more useful than others. The following query returns a subset of the columns and shows all those locks that are in a WAIT state:

```
SELECT
    TL.resource_type,
    DB_NAME(TL.resource_database_id) as DatabaseName,
    TL.resource_associated_entity_id,
    TL.request_session_id,
    TL.request_mode,
    TL.request_status
FROM sys.dm_tran_locks AS TL
WHERE TL.request_status = 'WAIT'
ORDER BY DatabaseName, TL.request_session_id ASC;
```

It would be more convenient to see just the waiting locks and the granted locks that they are waiting on. When a requested lock is waiting, it will be waiting on the same resource that the blocking process has already locked. The sys.dm_tran_locks DMV uses both the resource_associated_entity_id along with the resource_description to identify the locked resource, so we just need to use that as the method for joining the view with itself to pick out each waiting and granted lock for each given resource. The following query shows how you can do it:

Code View:

```
SELECT
    TL1.resource_type,
    DB_NAME(TL1.resource_database_id) AS DatabaseName,
    TL1.resource_associated_entity_id,
    TL1.request_session_id,
    TL1.request_mode,
    TL1.request_status
FROM sys.dm_tran_locks as TL1
JOIN sys.dm_tran_locks as TL2
ON TL1.resource_associated_entity_id = TL2.resource_associated_entity_id
AND TL1.request_status <> TL2.request_status
AND (TL1.resource_description = TL2.resource_description
OR (TL1.resource_description IS NULL AND TL2.resource_description IS NULL))
ORDER BY TL1.request_status ASC;
```

In the above query, you join the sys.dm_tran_locks view with itself and filter all those locks with a different request status (that picks out the GRANT and WAIT statuses) but with the same resource_associated_entity_id (a table, for example) and the same resource_description, or both with resource_description NULL (to cover cases in which there is no resource_description).

Output (two rows, wrapped):

Code View:

resource_type	DatabaseName	resource_associated_entity_id
KEY	AdventureWorks	72057594044022784
KEY	AdventureWorks	72057594044022784
request_session_id	request_mode	request_status
-----	-----	-----
53	X	GRANT
57	S	WAIT

Notice that the sys.dm_tran_locks DMV contains the database id, resource type, and locking information that is not available directly inside the sys.dm_os_waiting_tasks DMV. On the other hand, the sys.dm_tran_locks DMV does not contain the duration of the waiting time. Therefore it's natural to suggest combining the DMVs.

You might want to also query the view and return the actual object represented by the resource_associated_entity_id, whether it be a RID, key, page, or table.

Code View:

```

SELECT
    TL1.resource_type,
    DB_NAME(TL1.resource_database_id) AS DatabaseName,
    CASE TL1.resource_type
        WHEN 'OBJECT' THEN OBJECT_NAME(TL1.resource_associated_entity_id,
            TL1.resource_database_id)
        WHEN 'DATABASE' THEN 'DATABASE'
        ELSE
            CASE
                WHEN TL1.resource_database_id = DB_ID() THEN
                    (SELECT OBJECT_NAME(object_id, TL1.resource_database_id)
                     FROM sys.partitions
                     WHERE hobt_id = TL1.resource_associated_entity_id)
                ELSE NULL
            END
    END AS ObjectName,
    TL1.resource_description,
    TL1.request_session_id,
    TL1.request_mode,
    TL1.request_status
FROM sys.dm_tran_locks AS TL1
JOIN sys.dm_tran_locks AS TL2
ON TL1.resource_associated_entity_id = TL2.resource_associated_entity_id
WHERE TL1.request_status <> TL2.request_status
    AND (TL1.resource_description = TL2.resource_description
        OR (TL1.resource_description IS NULL
            AND TL2.resource_description IS NULL))
ORDER BY TL1.resource_database_id,
    TL1.resource_associated_entity_id,
    TL1.request_status ASC;

```

The previous version decodes the resource_associated_entity_id by looking it up in the sys.partitions catalog view. The subquery passes the found object_id value to the OBJECT_NAME() function. However, because the sys.partitions catalog view reports data per database, the inner CASE expression limits the subquery to returning values for just the current database when the resource_associated_entity_id is not DATABASE or OBJECT. You will have to run this query in each database in question to get all the object names.

The sys.dm_os_waiting_tasks DMV usually shows more information about the waiting task than the blocking task. Normally you can see the blocking_session_id, but often the blocking_task_address will be NULL because it is no longer an executing query. Even if it is not NULL, there is no guarantee that the task address will map to the query that originated the lock; it may just map to a later query in the transaction. As a result, it is not possible to get as much information about the blocking task as for the waiting task.

The Blocked Process Report

Another very useful way to find blocking is to use SQL Trace's Blocked Process Report. You can automatically trigger an event when a process has been blocked for greater than a specified amount of time. First, you use the sp_configure command to set the advanced option blocked process threshold to a value, let's say 60 seconds:

```
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
GO
EXEC sp_configure 'blocked process threshold', 60;
RECONFIGURE;
```

Then you can start a SQL Trace using the Profiler tool, choosing the Blocked process report event class in the Errors and Warnings group.

When you set up the trace, it's important to choose the relevant columns. SQL Server 2005's Books Online entry for "Blocked Process Report Event Class" explains all the data columns for the event class. Make sure you choose the TextData column so that you can inspect the report.

The event will fire when a blocked process is detected by the SQL Trace polling process. The TextData column blocked process report will return an XML-formatted set of data, which is actually quite readable. The report includes the spids involved in the blocking, and the input buffers for each, so you can see the commands. Data for the blocked process is shown first, and then the blocking process. [Figure 6-1](#) shows the blocked-process node of the XML output of the Blocked Process Report in XML Notepad.

Figure 6-1. A sample blocked-process node of the Blocked Process Report XML output using XML Notepad

[\[View full size image\]](#)

Tree View XSL Output

```

blocked-process
  process
    id
    taskpriority
    logused
    waitresource
    waittime
    ownerId
    transactionname
    lasttranstarted
    XDES
    lockMode
    schedulerid
    kpid
    status
    spid
    sbid
    ecid
    priority
    transcount
    lastbatchstarted
    lastbatchcompleted
    clientapp
    hostname
    hostpid
    loginname
    isolationlevel
    xactid
    currentdb
    lockTimeout
    clientoption1
    clientoption2
    executionStack

```

Select * from Sales.SalesOrderDetail

In [Figure 6-1](#), the inputbuf node shows you the blocked command, and the blocked-process node shows you, among other things, the spid, transaction level (transcount), isolation level, and database id of the blocked spid.

[Figure 6-2](#) illustrates the blocking-process node from the same XML file, also in XML Notepad.

Figure 6-2. A sample blocking-process node of the Blocked Process Report XML output using XML Notepad

[[View full size image](#)]

Tree View XSL Output

```

xml
  blocked-process-report
    blocked-process
      blocking-process
        process
          status
          spid
          sbid
          ecid
          priority
          transcount
          lastbatchstarted
          lastbatchcompleted
          clientapp
          hostname
          hostpid
          loginname
          isolationlevel
          xactid
          currentdb
          lockTimeout
          clientoption1
          clientoption2
          executionStack
            inputbuf
              #text

```

BEGIN TRANSACTION
 UPDATE Sales.SalesOrderDetail
 SET ModifiedDate = ModifiedDate + 1
 WHERE SalesOrderID = 43659

In Figure 6-2, the inputbuf node shows you the blocking command, and the blocking-process node shows you the spid, transaction level (transcount), isolation level, and database id of the blocking spid.

The benefit of the Blocked Process Report is that you have the blocking events recorded on disk in a trace file, along with the time and duration of the blocking. If your system has a lot of blocking, the report could get rather large. But you can increase the threshold option to narrow the size down to the just the longest ones.

The SQLDiag Utility

The SQLDiag utility has been enhanced for SQL Server 2005 and gives you a lot of information about your current system. You can run it as an executable from the command line, or you can set it up to run periodically as a service. You can read the output files directly, and you can also download the free SQLNexus utility to get reports for waits and blocking from <http://www.SQLNexus.net>.

You can also use the Microsoft PSS PerfStats collection of scripts in combination with SQLDiag to get blocking information. You can download the scripts and unzip them from the Microsoft PSS site at <http://blogs.msdn.com/psssql/archive/2007/02/21/sql-server-2005-performance-statistics-script.aspx>.

To get a sense of the data that SQLDiag with Perfstats collects, you can inspect the sp_perf_stats09.sql script. This script is used to create a stored procedure with the same name in tempdb, along with other stored procedures used by SQLDiag. The scripts also include sample command line executions of SQLDiag, with and without SQL Trace. For example, the command file StartSQLDiagNoTrace.cmd will run SQLDiag without running SQL Trace. You can then inspect the results in the SQLDiagOutput. The output of sp_perf_stats09.sql is in the file <Server>_sp_perf_stats09_Startup.OUT, and it contains the results of running the stored procedure. In the results, look for -- headblockersummary -- and you'll find the results for blocking.

Finding the Cause of Blocking

Lock-based blocking is caused by processes contending for locked resources. Just knowing the locks is not enough: to get to the root of the issue, you need to know the queries involved. Once you have identified the session id or spid, you can then track down the queries. SQL Server 2005 allows you to get at the actual task that is blocked, which is more information than we've been able to get in earlier versions.

Using the Blocked Process Report

The XML output in the TextData column for the Blocked Process Report will show the query text for both the blocking and the blocked sessions. By far this will be the most convenient method of finding the queries, but it may not always be convenient to use SQL Trace. In that case, you can join the sys.dm_os_waiting_tasks and sys.dm_tran_locks DMVs, and with some extra work, extract the queries associated with each.

Joining and

We'll tackle this in two steps. First, we'll build a join of the two DMVs to get the best information from each. Then we'll add subqueries to extract the query text. You can join on the waiting task's resource_address from sys.dm_os_waiting_tasks, and the lock_owner_address in sys.dm_tran_locks. Here's an example:

```
SELECT
    WT.session_id AS waiting_session_id,
    DB_NAME(TL.resource_database_id) AS DatabaseName,
```

```

WT.wait_duration_ms,
WT.waiting_task_address,
TL.request_mode,
TL.resource_type,
TL.resource_associated_entity_id,
TL.resource_description AS lock_resource_description,
WT.wait_type,
WT.blocking_session_id,
WT.resource_description AS blocking_resource_description
FROM sys.dm_os_waiting_tasks AS WT
JOIN sys.dm_tran_locks AS TL
ON WT.resource_address = TL.lock_owner_address
WHERE WT.wait_duration_ms > 5000
AND WT.session_id > 50;

```

In the above query, the results are paired, using sys.dm_os_waiting_tasks as the table: the waiting session is listed with its task information, and then the locking information for the waiting task from sys.dm_tran_locks is added. The last columns contain information about the blocking session, again from sys.dm_os_waiting_tasks.

Now for the second step: getting the actual query text for each session. By adding a subquery that joins the sys.dm_exec_requests DMV and the sys.dm_exec_sql_text() DMF, correlating back to the joining on the waiting task's session_id, you can extract the query text. Here's an example, adapted from a similar query in "["A Performance Troubleshooting Methodology." Chapter 1](#) of this volume.

Code View:

```

SELECT
    WT.session_id AS waiting_session_id,
    DB_NAME(TL.resource_database_id) AS DatabaseName,
    WT.wait_duration_ms,
    WT.waiting_task_address,
    TL.request_mode,
    (SELECT SUBSTRING(ST.text, (ER.statement_start_offset/2) + 1,
        ((CASE ER.statement_end_offset
            WHEN -1 THEN DATALENGTH(ST.text)
            ELSE ER.statement_end_offset
        END - ER.statement_start_offset)/2) + 1)
    FROM sys.dm_exec_requests AS ER
    CROSS APPLY sys.dm_exec_sql_text(ER.sql_handle) AS ST
    WHERE ER.session_id = TL.request_session_id)
        AS waiting_query_text,
    TL.resource_type,
    TL.resource_associated_entity_id,
    WT.wait_type,
    WT.blocking_session_id,
    WT.resource_description AS blocking_resource_description,
    CASE WHEN WT.blocking_session_id > 0 THEN
        (SELECT ST2.text FROM sys.sysprocesses AS SP
            CROSS APPLY sys.dm_exec_sql_text(SP.sql_handle) AS ST2
            WHERE SP.spid = WT.blocking_session_id)
        ELSE NULL
    END AS blocking_query_text
FROM sys.dm_os_waiting_tasks AS WT
JOIN sys.dm_tran_locks AS TL
ON WT.resource_address = TL.lock_owner_address
WHERE WT.wait_duration_ms > 5000
AND WT.session_id > 50;

```

Notice the two subqueries that retrieve the query text for the blocked and blocking queries. The first subquery that retrieves the query text associated with the waiting task uses the same logic that SQL Server 2005 Books Online describes in documenting sys.dm_exec_sql_text. By using a SUBSTRING function with the statement_starting_offset and statement_end_offset values, the query returns the exact query text associated with the waiting task.

To get the query for the blocking task, you need to first check whether the blocking_session_id is not NULL and not a negative number. (There are three codes that may be associated with negative blocking_session_id numbers that are discussed earlier in this chapter.) If the blocking_session_id is a positive value, you can still use the sql_handle but now from the sysprocesses view, which will return the entire batch.

Once you know the queries involved, the locked resource, and the type of locks involved, you can address the issue of how to resolve the blocking.

Resolving Blocking Problems

As mentioned previously, when you find lock-based blocking problems, the blocking may be caused by writers blocking writers, readers blocking writers, or writers blocking readers. Each type of blocking has its own potential solutions.

Killing a Session

The easiest but least satisfying method of resolving a blocking situation is to kill one of the sessions or spids using the KILL command. Sometimes this is the best solution in an emergency, and it is not uncommon to find someone who has submitted an ad hoc query on a production system that really should not be running and you can kill it safely.

But if the queries are all legitimate, killing one of the sessions might cause unexpected harm to the applications accessing the database. When the blocking is among queries, all of which are legitimate, you need to look deeper for a longer-term solution by looking at the cause of the blocking. If a particular blocking process is a long-running update or delete process, killing the session will cause a transaction rollback and the locks will not be released until the rollback is finished, and killing a spid in that context will not relieve the blocking until the rollback is complete.

Resolving Writer/Writer Blocking

There are not a lot of options for resolving writer/writer blocking in SQL Server because exclusive locks are always required by sessions in order to change data. When two sessions both need to change the same data, and therefore both sessions need to acquire exclusive locks on the same resource, you may need to rewrite the transactions or change the way they are run if you want to avoid the blocking problems. Here are some suggestions:

- Make write transactions shorter. One of the most effective methods for resolving writers blocking writers is to make transactions shorter. Exclusive locks acquired in a transaction are held until the transaction ends, so by reducing the duration of the transaction, you also reduce the time that the exclusive locks are required.
- Reduce the number of locks taken by writers. You can also use the same techniques you would use to reduce the number of locks. For example, you might reduce the number of rows changed by a mass update process, so that fewer numbers of rows are changed (inserted, updated, or deleted) at a given time, so that fewer rows need to be locked, or to reduce or remove lock escalation. You might also try to separate the contending writers by running them at different scheduled times, such as moving bulk load operations to a period with low system usage.

Resolving Reader/Writer Blocking

The number of options available for resolving reader/writer blocking is greater in SQL Server, primarily because you can adjust the isolation level of the transactions involved and reduce or remove the shared locks required by readers. Here are some suggestions:

- Lower the isolation level to READ UNCOMMITTED. One of the most common methods of resolving reader/writer blocking is to lower the isolation level, either by setting the reader's isolation level to READ UNCOMMITTED or placing a NOLOCK hint on the reader's query. This will cause SELECT statements to read uncommitted data. Until SQL Server 2005, this has been the only effective method for resolving reader/writer blocking, but it presents some risks.

First of all, the NOLOCK/READ UNCOMMITTED benefit of not taking any shared locks for a SELECT statement comes with the risk of reading uncommitted data. In a system in which few or no transactions are ever rolled back, that may mean the risk is very low. Even then, your query might read a newly inserted row from a header table but not see any of the detail table rows that have not yet been inserted. For critical queries that must return accurate aggregations or calculations based on consistent committed data, reading uncommitted data may not be acceptable.

Also, there is a small risk that a SELECT statement using a NOLOCK hint or READ UNCOMMITTED isolation level can fail. This can occur if SQL Server reads a page that has been deleted and therefore lacks the proper links to finish traversing the linked list and finish reading the required pages. When this occurs, you'll see error 601, and the query will abort. The text of error 601 is:

```
Could not continue scan with NOLOCK due to data movement.
```

It is not common to see this error, but if your applications use NOLOCK hints for READ UNCOMMITTED, they should test for it and resubmit the query if it occurs.

That's not the only problem with NOLOCK. As pointed out in Books Online (see the topic "Missing an updated row or seeing an updated row twice"), and also by Lubor Kollar (in his SQL Server Customer Advisory Team blog), it is possible for SELECT statements using the NOLOCK hint or READ UNCOMMITTED isolation level to skip rows that are the result of page splits that occur while the SELECT is underway. This can occur when the query optimizer chooses an allocation scan to scan a table for a SELECT statement, and page splits occur that put new pages into an earlier part of the allocation table. The SELECT statement only reads the allocation table forward, and so will miss such pages. Conversely, the SELECT statement may read rows twice due to page splits. (See the reference in the bibliography, "[Additional Resources and References](#)," to the blog by Lubor Kollar.)

- Check for the correct isolation level You may also see transactions using the REPEATABLE READ or SERIALIZABLE isolation level that do not require it. Lowering these isolation levels to the default READ COMMITTED will allow SQL Server to release shared locks before the transaction ends, keeping them for a shorter time. The Blocked Process Report XML output file's isolation-level column will tell you the isolation level of each transaction. Range locks involved in the blocking indicate the transaction is using the SERIALIZABLE isolation level. In some cases, developers may accidentally use this isolation level when they do not really need it.
- Use the row-versioning-based Snapshot Isolation levels In SQL Server 2005, you can avoid the problems associated with reading uncommitted data by using one or both of the snapshot-based isolation levels. By far the most effective method is to change the way the default READ COMMITTED isolation level works, and change the database by setting READ_COMMITTED_SNAPSHOT ON. This changes the way SELECT statements read committed data; instead of taking shared locks, they read prior versions of any data that was changed by transactions that began at the start of the SELECT statement execution, or later than it, for the duration of the SELECT statement. All DML transactions (ones that change data) will cause row versioning when this option is enabled, and all SELECT statements that run under the READ

COMMITTED isolation level will potentially read versioned data. The cost of this option is increased activity in tempdb, where the version store is located, as well as the cost of traversing pointers to retrieve the appropriate versions of particular rows. In addition, there are potential issues with READ COMMITTED SNAPSHOT and triggers, if those triggers rely on reading current data only, which are discussed in the "[Troubleshooting Row-Versioningâ€”Based Snapshot-based Isolation Levels](#)."

- Separate readers from writers For longer-term solutions, you may want to consider separating the contending reader queries from the writer queries entirely. Sometimes this is called "separating reads from writes," but that phrase oversimplifies matters because often only a subset of all the reader queries can be redirected to a read-only copy of the database. Often in even the most active OLTP databases, the amount of read activity greatly exceeds write activity. A majority of those reads may be able to read the data from another server or database. Creating a reporting server separate from the main server, and fed data by transactional replication (for example), may allow you to offload many of the contentious reader queries. Unfortunately those same queries may attempt to lock the same data that replication stored procedures are updating, so you may need to apply READ_COMMITTED_SNAPSHOT to the subscriber database in order to eliminate the contention.

Another method for separating readers from writers would be to create a database snapshot of the current database on the SQL Server instance, a snapshot that has data current as of a specified time. The SELECT statements run against a database snapshot (which is read-only) will not acquire as many shared locks as they would on a read-write database.

Sometimes the most contentious reader queries however are just those that need up-to-the-second data, so it's not possible to separate the contending readers from the writers. In that case, READ_COMMITTED_SNAPSHOT remains the best option.

Troubleshooting Deadlocking

A deadlock in SQL Server consists of two threads that block each other, and each is waiting on the other to finish. This type of mutual blocking is commonly called a cycle, because you have two blockers in a kind of inverse relationship. SQL Server's deadlock monitor process periodically checks for such cycles and chooses one of the tasks as a deadlock victim. The deadlock victim's session has its batch aborted, its transaction rolled back, and it receives the following 1205 error message template (from the master.dbo.sysmessages system table):

Code View:

```
Transaction (Process ID %d) was deadlocked on %.*ls resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```

SQL Server will insert the session id in the %d location, and write the type of resource deadlocked on, in the %.*ls position. The end result is that your victim session will see something like:

Code View:

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```

Error message 1205 is perhaps notable more for what it doesn't tell us than for what it does. It tells you the victim's process id and suggests you rerun the transaction. The problem here is that your process might not have been a transaction, because SELECT statements can be victims of deadlocks, too. Even more important is what error message 1205 doesn't say:

- The session id of the other process.
- What the conflicting queries were.
- What the conflicting resources were.

To find this information out, and resolve the deadlock, you need to use some additional tools that come with SQL Server 2005. (We assume you are familiar with how deadlocks can arise; for more information, see the "Deadlocks" section in Inside SQL Server 2005: The Storage Engine, Chapter 8.)

Types of Deadlocking

Deadlocking as detected by SQL Server always involves some kind of threads or processes that conflict over resources. The types of conflicts depend on the types of resources and the types of incompatibility.

Potential Deadlocks

The types of resources that can deadlock include:

- Locks: Locks on database objects, including data rows, pages, tables, and index keys.
- Worker threads: This can include scheduler and CLR synchronization objects.
- Memory: In which two processes each need more memory but must wait for the other before it can be granted.
- Parallel threads within a single query.
- Multiple Active Results Sets (MARS) resources: Internal conflicts within MARS threads.

By far the most common deadlocks are lock-based, and that type is what we will focus on in this chapter.

The other types of deadlocks are more specialized and generally much less frequent. Deadlocks involving parallelism are resolved internally by SQL Server and the query will finish. A high frequency might warrant lowering the max degree of parallelism for the queries. (For scheduler-based deadlocks, see the reference to Bob Dorr, "How to Diagnose and Correct Errors 17883, 17884, 17887, and 17888" in the bibliography, "[Additional Resources and References](#)")

Note



Deadlocking cannot occur among latches. SQL Server 2005 includes special code to prevent latches from forming the kind of blocking cycle that can cause a deadlock. (For more information, see Inside SQL Server 2005: The Storage Engine, Chapter 8, "Locking and Concurrency".)

The SQL Server deadlock monitor also can detect deadlocking within SQL CLR managed code, but it will not automatically release resources if the managed code is chosen as the deadlock victim. It is important that the code properly handle deadlock exceptions and either retry or release the resources. For a treatment of CLR deadlocks, see the reference to Joe Duffy, "Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps" in the bibliography, "[Additional Resources and References](#)".

Lock-Based Deadlocking

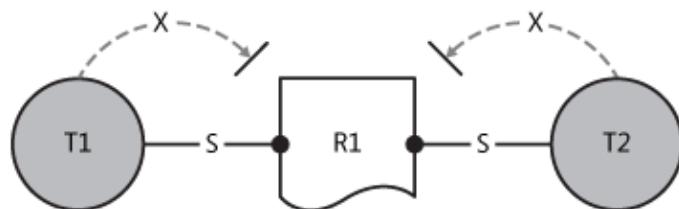
Deadlocking is all about timing and requires just the right type of blocking. Two processes must be trying to gain access to the same resources at almost the same time for blocking and then deadlocking to occur. That means that the locking activity of the processes must overlap in time.

Lock-based deadlocking also requires at least one transaction, because at least one exclusive lock is usually present to initiate some blocking that will result in two tasks blocking each other. The transaction might be an explicit transaction that a user submits, or the autocommit type of transaction that SQL Server performs with a single INSERT, UPDATE, or DELETE. The chance of seeing lock-based deadlocks grows with the amount of activity on your system, because the probability of simultaneous conflicts also rises. Also, the longer your transactions run, the greater the chance of deadlocks, because again the probability of transactions overlapping in time increases.

Just as with blocking, it is convenient to think of lock-based deadlocking in terms of writers (requesting or holding exclusive locks or intent exclusive locks) and readers (requesting or holding shared or intent shared locks). Let's take a look at the types of deadlocking in terms of readers and writers.

1. Conversion deadlock. The simplest deadlock over a single resource is relatively rare. Normally deadlocks involve a conflict over two or more resources, but it is possible for two threads to conflict over one resource, and this is called a conversion deadlock. [Figure 6-3](#) illustrates this type of deadlock.

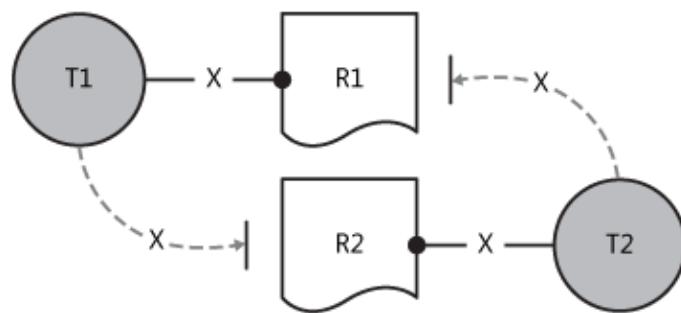
Figure 6-3. A conversion deadlock involving one resource



A conversion deadlock starts with both threads T1 and T2 obtaining a lock on the single resource R1. The only type of locks that are compatible in this fashion are shared locks. Then each tries to convert its own shared lock to a type of lock that is incompatible, such as an exclusive lock, and the deadlock results. Conversion deadlocks are rare because it is not often that you would need to convert locks from shared to exclusive, but they are possible. Also, using the default READ COMMITTED isolation level, the two initial reads will release their shared locks as the data is read. In order, then, to keep the shared locks active so they can be converted, you have to raise the isolation level. Because of that, such a conversion deadlock is not possible using the default isolation level, another reason it is relatively rare.

To resolve a conversion deadlock, you can strengthen the initial SELECT statements to use the UPDLOCK hint, which will force one of the transactions to wait until the other finishes.

2. Writer/writer cycle deadlock. A cycle deadlock involves two resources, where each thread has some kind of lock on its own resource and the other thread requests some kind of incompatible lock. The two threads form a cycle, each wanting an incompatible lock on the other's resource, and neither is able to finish. The simplest form of a cycle deadlock, and perhaps the clearest to understand, involves just writers. [Figure 6-4](#) shows the general structure of a writer/writer cycle deadlock.

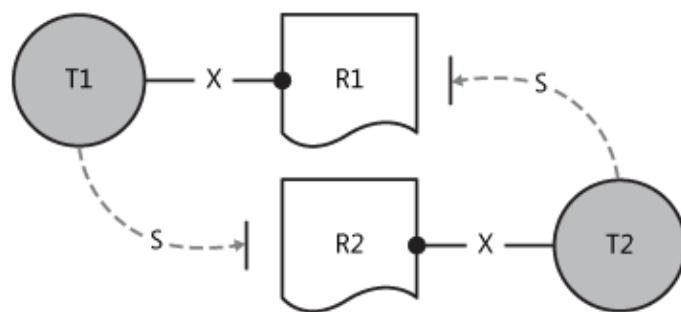
Figure 6-4. A writer/writer cycle deadlock

In order for this deadlock to occur, each thread must be in a transaction and must access the resources in the opposite order. In other words, the sequence is:

- ◆ Step 1. T1 acquires an exclusive lock on R1.
- ◆ Step 2. T2 acquires an exclusive lock on R2.
- ◆ Step 3. T1 requests an exclusive lock on R2.
- ◆ Step 4. T2 requests an exclusive lock on R1.
- ◆ Step 5. T1 moves from R1 to R2, while T2 moves from R2 to R1.

The general solution for this type of deadlock is to access the resources in the same order. Often this can be solved by using stored procedures: If T1 and T2 had to do their work using the same stored procedure, the order of accessing the resources would be the same.

3. Reader/writer cycle deadlocks. More often, deadlocks involve a combination of writer and reader activity, with both exclusive and shared locks involved. [Figure 6-5](#) shows two threads attempting to read each other's data before they are done with it.

Figure 6-5. A simple reader/writer cycle deadlock

In this type of deadlock, each thread must attempt to read the other's data before it finishes its transaction. The sequence is as follows:

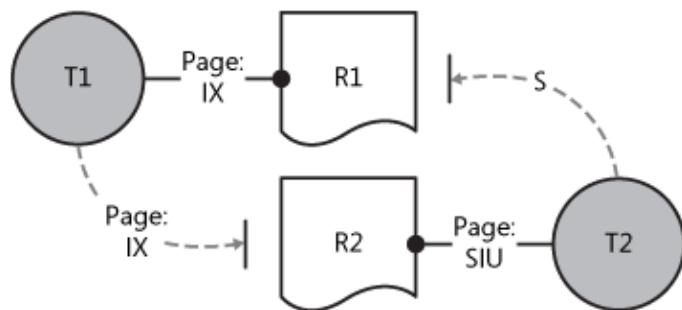
- ◆ Step 1. T1 acquires an exclusive lock on R1.
- ◆ Step 2. T2 acquires an exclusive lock on R2.
- ◆ Step 3. T1 requests a shared lock on R2.

- ◆ Step 4. T2 requests a shared lock on R1.

[Figure 6-5](#) shows a simple form of a reader/writer deadlock, in which the lock incompatibility results from a straightforward conflict between exclusive and shared locks. The shared lock may be acquired directly, through a clustered index, or indirectly, through a nonclustered index, but in either case, the mutual blocking results in a deadlock.

The types of reader/writer deadlocks vary considerably, because the number of possible lock incompatibilities is large. Illustrations with just a shared and exclusive lock on a resource can be a little misleading because often more complex locks can be involved. For example, [Figure 6-6](#) illustrates a deadlock that involved a wider mix of locks and incompatibilities.

Figure 6-6. A cycle deadlock involving shared locking



In this example of a cycle deadlock, the following steps occur:

- ◆ Step 1. T1 is an update statement that gets an exclusive lock on a row in the data page R1.
- ◆ Step 2. T2 is a different update statement that will update a row on page R2 and acquires an SIU lock.
- ◆ Step 3. In the meantime, T1 also wants to extend its update to a row on page R2 and attempts to get an IX lock first on the page, which is incompatible with the SIU lock.
- ◆ Step 4. Then T2 attempts to get a shared lock on the page R1 and is blocked.

This type of deadlock requires a closer analysis of the statements involved.

4. Cascading deadlocks. In SQL Server 2000, sometimes you could see more than two processes involved in a deadlock in the output of the 1204 trace flag. In those cases, SQL Server would choose one victim and remove one deadlock, but the remaining processes could also continue to be deadlocked. In such cases, victims would be chosen one by one until all the deadlocks are removed. SQL Server 2005 also shows cascading deadlocks in the 1223 trace flag.

Troubleshooting deadlocking, just as with troubleshooting blocking, requires that you first detect them, analyze to find the cause, and then perform some action to resolve them.

Detecting Deadlocks

Just as with blocking, SQL Server 2005 gives you a number of tools for detecting deadlocking on your system.

Quickly Detecting Deadlocking

To get a quick overview of potential deadlocking on an active SQL Server instance, you can use the Number of Deadlocks/Sec counter in the SQLServer:Locks group of System Monitor (Perfmon). You can drill down to the type of deadlock using the same "instances" choices that you get for the rest of the locks counters.

Using the System Monitor SQLServer:Locks Number of Deadlocks/Sec counter can be very useful in a test environment in which you can simulate multiuser behavior. The counter can help you isolate the parts of the application that will deadlock, and under what conditions, so that you can investigate later to get more specific information.

Also, when a database is in production and you are getting reports of deadlocking, it can be useful to use the Perfmon counters to determine the frequency of deadlocking and the type based on the more granular information you can get from the Number of Deadlocks/Sec counter. But the counter can only give you aggregate information, and at some point you will need more detailed information.

The SQL Trace Deadlock Event Classes

You can also use the SQL Trace deadlock events. You can still use the Lock:Deadlock and Lock:Deadlock Chain events in SQL Server 2005's SQL Trace. These event classes will show that a deadlock has occurred, and between which spids. If you also capture the SQL:StmtStarting and SQL:StmtCompleted event classes, you can link the spid ids to find out the statements they were trying to execute at the time of the deadlock detection. You can infer the victim spid because it will have a starting command and no ending command.

For better and more graphical information, however, SQL Server 2005 also gives you the Deadlock graph event class which you can use to help determine the overall shape of the deadlock, but the graph does not expose the queries present for each process when the deadlock occurred. If you save the XML output of the deadlock event to a file, you will see the same output as the 1222 trace flag. The resulting .XDL file is in a compliant XML format, so you can read it in a utility like XML Notepad, something you cannot do with the 1222 trace flag output.

Trace Flags 1222 and 1204

For a fully detailed report about a deadlock, SQL Server 2005 provides you with the 1222 trace flag. You can set this interactively by executing

```
DBCC TRACEON (3605, 1222, -1)
```

The DBCC TRACEON command allows multiple flags to be set, and in this case you need two: 3605 directs output to the SQL Server Error Log, and 1222 captures deadlock information. In SQL Server 2005, you need the -1 switch at the end to indicate that you want the data collection done server-wide and not just for your session. The trace flag 1222 is the best tool for getting to the cause of the deadlock, because it shows the statements being executed by each task at the time of the deadlock. That brings us to our next topic.

Determining the Cause of Deadlocks

To get to the cause of a deadlock, you need to know not only the resources and the types of locks involved, but also what initial locks led to the deadlock. For this kind of information, you need the 1222 trace flag. It will show you the commands being executed by each process at the time of the deadlock, the commands that were blocked and formed the conflict or cycle that led to the deadlock. Finding out the statements that took

the initial locks on the resources that form the base of the conflict requires knowledge of the history of each task and its transactions, and is not revealed by the 1222 trace flag. For that information, you need the complete history of the transactions, which requires that you be running a SQL Trace that captures the history for each of the spids involved in the deadlock.

In SQL Server 2000 and earlier, the only trace flag for detecting deadlocks was the 1204 flag, and it can still be used in SQL Server 2005. Why not continue to use it? There are two reasons that the 1222 flag is superior to 1204.

First, the 1204 flag's output has been updated to report the longer hobtid (heap or B-Tree ID) format. For example, in SQL Server 2000 a locked key resource would have been reported in 1204 in a db_id:object_id:index_id format, such as KEY: 2:1977058079:1. You could use the DATABASE_NAME() and OBJECT_NAME() function to find out the database and object name corresponding to the first two numbers, and then access the sysindexes table to determine the index (and index id of 1 indicates the clustered index of the table). Analogous formats exist for RID, table, page, and extent formats in SQL Server 2000. However, in SQL Server 2005, the 1204 trace flag reports its resources in terms of the hobtid (heap or B-Tree ID). This change makes the 1204 output a little less readable (though it's not that readable to begin with).

The second reason to prefer the 1222 trace flag is that it decodes the hobtid for you anyway, returning the names of the database, objects, and indexes involved in the deadlock. Reading the output of the trace flag takes some getting used to, but after a while it becomes second nature. The following sections take apart a trace of a simple deadlock in which two processes each start a transaction, lock a resource of their own, and then try to read the other's resource.

The output of the 1222 trace flag is a quasi-XML format, whereas the 1204 output is not in any standard output. Neither output is particularly easy to read, but they both have patterns that you can get used to.

Note



When you use the 1222 or 1204 trace flags, output will be sent to the SQL Server Error Log. You can use the SQL Server Management Studio Log Viewer to view the information, but the output is sorted starting with the most recent, so it's basically upside down. (In SQL Server 2000's Enterprise Manager, the 1204 output is actually garbled and unreadable.) The best way to read the output of either trace flag is to copy the error log somewhere else and then read it using Notepad. You can then trim out extraneous information, as well as time and date information, to isolate the output.

To determine whether deadlocks are occurring based on either the 1222 or 1204 trace flags, you must monitor the SQL Server Error Log, or use a third-party tool that will monitor it for you. SQL Server has no built-in alerting based on this output, so in the absence of any other tool, you have to periodically inspect the error log. You can read the trace flag 1222 output in the Error Log, or export it to a text file and remove the prepended timestamp and spid information to make it more readable.

The 1222 output is in an XML-like format but does not comply with any XSD schema, so it cannot be read using a utility-like XML Notepad. The trace flag 1222 XML output can be divided into three sections: the deadlock victim, the participant processes, and the resources.

1. Deadlock Victim. The first section of the 1222 output identifies the victim of the deadlock by process name. This replaces the Node numbers used by the 1204 trace flag:

```
deadlock-list =processesb5588
```

2. Processes. The next section identifies the victim and survivor processes. The first process listed here

is the victim, which you can tell by the process id, but it might just as well have been the other process; the first process might be the survivor, not the victim.

Code View:

```
process-list
  process id=processeb5588 taskpriority=0 logused=356 =KEY:
8:72057594043236352 (2e0032505f99) waittime=4484 ownerId=103932
transactionname=user_transaction lasttranstarted=2007-04-02T22:39:29.687
XDES=0x82f66be0 lockMode=X schedulerid=2 kpid=2232 status=suspended sbid=0
ecid=0 priority=0 transcount=2 lastbatchstarted=2007-04-02T22:39:38.623
lastbatchcompleted=2007-04-02T22:39:29.687 clientapp=Microsoft SQL Server Management
Studio - Query hostname=MAUI hostpid=996 loginname=MAUI\SQLAdmin =read
committed (2) xactid=103932 locktimeout=4294967295 clientoption1=671090784
clientoption2=390200
  executionStack
    frame procname=adhoc line=2 stmtstart=60
sqlhandle=0x02000000b6529c1c23f04818a690de103bc008f6e338b312
UPDATE [Person].[Contact] set [Title] = @1 WHERE [ContactID]=@2
    frame procname=adhoc line=2 stmtstart=92
=0x0200000096ccc627782a71dd2eaf00956644a1a9ca4e1b9
Update Person.Contact
Set Title = 'Mrs.' Where ContactID = 1070

-- Start the transaction
Update Person.Contact
Set Title = 'Mrs.' Where ContactID = 1070
```

For the first process, processeb5588, there is a wealth of information, some of which you would expect, such as the wait resource, database id, the spid number, and the input buffer (also provided by the 1204 flag). But we get new information about the transaction isolation level and the sql handle of the command that actually acquired the resource.

The next process listed has a somewhat similar output:

Code View:

```
process id=processeb56d8 taskpriority=0 logused=480 =KEY:
8:72057594058899456 (07005a186c43) waittime=10000 ownerId=103907
transactionname=user_transaction lasttranstarted=2007-04-02T22:39:25.543
XDES=0x83182a40 lockMode=U schedulerid=2 kpid=2224 status=suspended spid=54 sbid=0
ecid=0 priority=0 transcount=2 lastbatchstarted=2007-04-02T22:39:33.107
lastbatchcompleted=2007-04-02T22:39:25.543 lastattention=2007-04-02T22:23:43.903
clientapp=Microsoft SQL Server Management Studio - Query hostname=MAUI hostpid=996
loginname=MAUI\SQLAdmin =read committed (2) xactid=103907 currentdb=8
locktimeout=4294967295 clientoption1=671090784 clientoption2=390200
  executionStack
    frame procname=adhoc line=2 stmtstart=158
=0x020000008ecb8f2871e589c24838bceea42dbb37ef32bf27
WHERE [ContactID]=@2
    frame procname=adhoc line=2 stmtstart=158
sqlhandle=0x020000002b416c38c09e56211be28c59eed9e4919bbd7adc
Update HumanResources.Employee
Set SalariedFlag = 1
Where ContactID = 1070

-- Other transaction
Update HumanResources.Employee
Set SalariedFlag = 1
Where ContactID = 1070
```

3. Deadlocked resources. Last, the 1222 output lists the resources involved in the deadlock. This is the

most readable portion of the output and probably where it's best to focus.

```

resource-list
hobtid=72057594058899456 dbid=8
=AdventureWorks.HumanResources.Employee
=PK_Employee_EmployeeID id=lock8012fc00 =X
associatedObjectId=72057594058899456
    owner-list
    =processeb5588
    waiter-list
    =processeb56d8 requestType=wait
        keylock hobtid=72057594043236352 dbid=8
        objectname=AdventureWorks.Person.Contact indexname=PK>Contact>ContactID
        id=lock80130780 =X associatedObjectId=72057594043236352
    owner-list
    =processeb56d8
    waiter-list
    =processeb5588 requestType=wait

```

The 1222 output decodes the hobtid for you, saving a very tedious step that the 1204 flag still requires you to do. In this case, we happen to know that both indexes are clustered primary keys, so we know that a keylock refers to a row of the table. At a glance, you can tell that one process had an exclusive lock on a row in the AdventureWorks.HumanResources.Employee table, and the other process was waiting to get an update lock. At the same time, the other process had an exclusive lock on a row in the AdventureWorks.Person.Contact table, and the first process is waiting to get an exclusive lock on it.

You can summarize the output into a simple table ([Table 6-2](#)), which then exposes most of the causes of the deadlock.

Table 6-2. A Sample Summary of the 1222 Trace Flag Output

Victim	Survivor	Process	id	process	ses	b5588	processeb5588	process	ses	b56d8	Database	AdventureWorks	AdventureWorks	Resource	keylock	keylock	Type	Resource	Hu
Granted	XXLock	Requested	UX	LastUpdate	Person	Contact	Update	HumanResources	Employee	CommandSet	Title	= 'Mrs.'	'Set SalariedFlag = 1 Where ContactID = 1070	Where	ContactID = 1070				

Once you know the resources and the statements that conflicted on them, you often have enough information to diagnose and resolve the deadlock. In some cases, you may need to dig deeper to find out what statements acquired the locks on the resources to begin with, which will require running SQL Trace to get a full history for each transaction.

Note



The output of the 1222 trace flag in the SQL Server error log can be voluminous. It can help to recycle the error log (using the system stored procedure `sp_cycle_errorlog`) periodically and then save the error logs elsewhere in order to isolate the deadlocks you wish to analyze.

Resolving Deadlocks

Once you know the statements and resources involved in a deadlock, you can use a number of strategies to resolve it. A good set of steps to follow is set out in SQL Server 2005's Books Online under the topic "Minimizing Deadlocks." In this section, we'll review those steps with some additional comments and add a

few other things you can check.

To start with, in order to really understand a deadlock, you need to use the 1222 trace flag. The 1222 trace flag will tell you both the types of locks involved in the deadlock as well as the ending statements that caused the deadlock.

General Strategies for Resolving Deadlocks

The following steps can be considered general guidelines for resolving any deadlock. In the subsequent sections, we look at more specific details for troubleshooting specific types of deadlock.

1. Determine whether a short-term or long-term solution is required. Sometimes you have to fix a deadlock in production and must find a quick solution. In such cases, you may be able to implement a minor change to a single statement, such as adding a NOLOCK hint to a SELECT statement. At other times, you may be able to look for a longer-term solution, by working through the transactions involved and testing changes to the way the code works.
2. Isolate the deadlocking code and reproduce it in a test setting. From the information given to you by the 1222 trace flag, you may be able to determine exactly what queries are involved in causing the deadlock, as well as locking the initial resources. A good example of this is when you see a stored procedure deadlocking with another instance of itself, or two different stored procedures deadlocking. In that case, you may be able to run the procedure code step-by-step in Management Studio and re-create the deadlock.

Other cases, such as deadlocks between ad hoc SQL queries, or between executions of the system stored procedure sp_executesql, will require more information to find out what statements are acquiring the locks to begin with, and what the transaction boundaries are. In those cases you must capture the SQL Trace events and then review them in Profiler, filtering on just the deadlocking spids, to find out exactly how the transactions occurred and which statements locked the resources that the latter statements deadlocked over.

3. Check the granularity and amount of locking in the deadlocks. The existence of any table locks could indicate lock escalation. You may also be able to reduce the amount of locking with better indexes. For example, updates to keys of clustered indexes will require subsequent locks of nonclustered index keys, which could lead to deadlocking.
4. Check for missing indexes. In some cases, a process may end up scanning more rows than necessary in order to perform an operation. For example, a SELECT statement may need to scan the rows in a table to retrieve a column that is not indexed. Or an UPDATE or DELETE may have a WHERE clause that references an unindexed column, and therefore must scan the table's rows to find all those qualifying for the change. If two processes access the table in a different order, they will block on each other's exclusive locks and may deadlock.

To help determine whether a different index might help, place the code for the transactions separately in SQL Server Management Studio and analyze the queries using the Database Engine Tuning Advisor, or use the missing_index dynamic management objects. If it recommends any additional indexes, add them and then retest the deadlocks. (For further discussion of this technique, see [Chapter 1](#), "A Performance Troubleshooting Methodology," and the blog "Deadlock Troubleshooting" by Bart Duncan in the bibliography, "[Additional Resources and References](#).")

5. Shorten the transactions. Because exclusive locks are held to the end of a transaction, the longer the transaction, the greater the opportunity exists for blocking and deadlocking. Rewriting the code involved may make it possible to make the transactions take less time, thereby reducing the opportunity for them to overlap in time. When you inspect the statements involved in the deadlock, you may find that the queries can be made more efficient or made to travel different paths. A DELETE statement may, for example, have a WHERE clause that can be changed to find its path more directly.

6. Retry the transaction if it is a deadlock victim. You can use the TRY/CATCH construct in SQL Server 2005 to trap a deadlock and then retry it a certain number of times. This may work best for infrequent deadlocks, where you can be confident that the deadlock is an anomaly and that the statements or transactions involved will only rarely run simultaneously. Here's a skeletal example of how you can rewrite your code to retry the transaction using Transact-SQL:

```

DECLARE @Tries tinyint, @Error int;
SET @Tries = 1;
WHILE @Tries <= 3
BEGIN
    BEGIN TRANSACTION;
    BEGIN TRY
        -- <code goes here>
        IF XACT_STATE() = 1 COMMIT;
        BREAK;
    END TRY
    BEGIN CATCH
        SET @Error = ERROR_NUMBER();
        IF @Error = 1205
        BEGIN
            IF XACT_STATE() = -1 ROLLBACK;
        END
        SET @Tries = @Tries + 1;
        CONTINUE;
    END CATCH;
END;

```

Notice that the BEGIN TRANSACTION statement is inside the WHILE loop, not before it. That's because if a deadlock error occurs, the transaction inside the CATCH block is uncommittable and you must roll it and then restart it. (For more about uncommittable transactions, see "Using TRY . . . CATCH in Transact-SQL" and "XACT_STATE" in SQL Server 2005 Books Online.) In the previous code, if a deadlock occurs, control passes to the CATCH block, where if the error number 1205 is found, a ROLLBACK occurs, ending the transaction. The CONTINUE command then passes control back to the WHILE loop for a further iteration if the value of @Tries is <= 3. If the BEGIN TRANSACTION were placed before the WHILE loop, it would not execute again and the code would no longer be within a transaction. You can then add a RAISERROR after the third iteration to notify the client that the deadlock retries failed.. (For more details on using TRY/CATCH, see the reference to Inside Microsoft SQL Server 2005: T-SQL Programming, Chapter 10, in the bibliography, "[Additional Resources and References](#)."

7. Lower the deadlock priority of one process. If the statements involved are different, that is, if they come from different procedures, and if one of them is clearly a lower priority than the other, you can use the SET statement to set the DEADLOCK_PRIORITY of one of them lower. SQL Server 2005 greatly extends the range of deadlock priorities that you can choose, beyond LOW and NORMAL, adding a HIGH option as well as a range of 21 numeric values ranging from -10 to 10. (See "SET DEADLOCK_PRIORITY" in SQL Server 2005 Books Online. For a detailed discussion about using DEADLOCK_PRIORITY, see Inside SQL Server 2005: The Storage Engine, Chapter 8.)

Setting deadlock priorities is not a satisfying solution because by itself, it does not remove a deadlock; it just allows you to specify a preferred victim. However, you may be able to combine DEADLOCK_PRIORITY usage with the retry capability of TRY/CATCH so that the lower-priority transactions, if they become victims, can be retried and then eventually succeed. (Note that when a task is being rolled back, it cannot be a victim of a deadlock, no matter what its deadlock priority is set.)

8. Use bound connections. If you have two transactions that must run simultaneously, but each must have its own connection, you can bind one of the transactions to the transaction context of another, and their locks will no longer be incompatible. They can also then read each other's changes, as well as write to the same objects. However, they will also be able to read each other's uncommitted data, so there are some risks involved. (For more details about bound connections, see Inside SQL Server

2005: The Storage Engine, Chapter 8.)

Resolving Writer/Writer Deadlocks

Deadlocks between writers tend to be a bit easier to resolve because only exclusive locks are involved. Here are the basic troubleshooting steps:

1. Access objects in the same order. If the queries that are deadlocking involve transactions accessing more than one table, make sure that the transactions are all accessing those tables in the same order. This may remove the deadlocking but actually increase blocking, because each transaction will have to wait for others to finish.
2. Run mass updates during off-hours. In some cases, such as archiving, you may need to perform mass updates or deletes that remove large numbers of rows. If you can move the mass updates to hours when other activity on the system is low, you can reduce the chance of deadlocking.

Resolving Reader/Writer Deadlocks

By far the most common and most subtle deadlocks involve both readers and writers, that is, they are some combination of shared and exclusive locks. Fortunately, SQL Server 2005 provides you with a number of ways to reduce or eliminate the locking due to reads. In reader/writer deadlocks, if you could remove the conflict caused by the read, the deadlock would be resolved.

1. Remove the locks caused by reads. You can also resolve a reader/writer deadlock by simply removing the shared locks entirely. SQL Server 2005 has two strategies for doing this: employing one of the Snapshot Isolation options, or lowering the isolation level to READ UNCOMMITTED.

The best method for removing shared locks is to use one of the Snapshot Isolation options available with SQL Server 2005. For reader/writer deadlocks that occur under the READ COMMITTED isolation level, just changing the database's option to READ COMMITTED SNAPSHOT will be enough to remove the deadlocks. No shared locks will be taken, and changes to database data will be versioned and kept in tempdb. With the READ COMMITTED SNAPSHOT option SET TO ON in a database, you are still guaranteed that unless you override it, SELECT statements will read only committed data, and will not take shared locks. The cost is the increase of activity in tempdb. (See the next section in this chapter, "[Troubleshooting Row-Versioningâ€”Based Snapshot-Based Isolation Levels in tempdb](#)" for more details.)

If you have transactions currently requiring REPEATABLE READ or SERIALIZABLE isolation levels, you may be able to replace them with the SNAPSHOT isolation level. Like READ COMMITTED SNAPSHOT, SELECT statements will not take shared locks and your transactions will read only committed data. The difference is that with the snapshot isolation level, your transaction read data that is consistent as of the beginning of the transaction until the transaction finishes. This can cause increased activity in tempdb, as well as make your transactions subject to update conflicts. (For troubleshooting update conflicts, see "[Troubleshooting Row-Versioningâ€”Based Snapshot-Based Isolation Levels](#)," in the next section.)

Note



Changing an isolation level only affects shared locks. Exclusive lock behavior is not affected by isolation-level options. Exclusive locks are always held to the end

of a transaction no matter what the isolation level is.

The READ UNCOMMITTED isolation level causes SQL Server to stop taking shared locks on data objects such as tables, rows, or pages, and on indexes. You can cause all SELECT statements in a session to execute using READ UNCOMMITTED by issuing the SET TRANSACTION ISOLATION LEVEL READ COMMITTED command. Alternatively, you can lower the isolation level of an individual SELECT statement to READ UNCOMMITTED on a table-by-table basis by using the NOLOCK hint. The benefit is that no shared locks are taken, reducing locking overhead, and eliminating the blocking of SELECT statements by exclusive locks, and also resolving reader/writer deadlocking. The cost is that your SELECT statements may read uncommitted data, and they may even fail under some rare conditions.

(For a fuller discussion of isolation levels, see Inside SQL Server 2005: The Storage Engine, Chapter 8. For a discussion of the problems with the NOLOCK table hint and the READ UNCOMMITTED isolation level, see "[Resolving Reader/Writer Blocking](#)" earlier in this chapter.)

2. Check the isolation level of the transactions. The 1222 trace flag will tell you the isolation level of each transaction. In addition, if you see any range locks involved in the deadlock, that indicates the transaction is using the SERIALIZABLE isolation level. In some cases, developers may accidentally use either the SERIALIZABLE or REPEATABLE READ isolation levels. Adjusting the isolation level back down to the default READ COMMITTED may help reduce or remove the deadlocking.

An example of where this strategy can work is the conversion deadlock. Suppose a stored procedure must read a row before updating it, and in order to make sure that the shared lock is not released, the developer set the isolation level to REPEATABLE READ or SERIALIZABLE. As a result, when two sessions run the procedure nearly simultaneously, they each attempt an update and form a deadlock. To resolve this, you can set the isolation level back to READ COMMITTED and use the HOLDLOCK and UPDLOCK table hints on just the one table; the first procedure to execute the SELECT statement will acquire and keep the UPDLOCK, and no other session will be able to change the data until this transaction finishes. If another instance of this procedure runs nearly simultaneously with this one, it will have to wait until the first transaction's update to the data is finished before it will be granted its UPDLOCK.

Troubleshooting Row-Versioningâ Based Snapshot-Based Isolation Levels

SQL Server 2005's snapshot isolation levels are new features of the database engine that are specifically meant to improve concurrency. These two levels are:

- READ COMMITTED SNAPSHOT: a variation of the default READ COMMITTED isolation level
- SNAPSHOT ISOLATION: a new isolation level for explicit transactions. (We'll refer to this option as Full SNAPSHOT to distinguish it more clearly from READ COMMITTED SNAPSHOT.)

We refer to these two options together as the snapshot-based isolation levels. A SELECT statement executing in a database with READ COMMITTED SNAPSHOT enabled will return committed data that is consistent as of the beginning of the SELECT statement. A SELECT statement in a transaction running in the Full SNAPSHOT isolation level, will return data that is consistent as of the beginning of the transaction. Note that for the purpose of data consistency, the transaction is assumed to start the first time any data is accessed.

In this section, I will assume you are familiar with both snapshot-based isolation levels. (See "Row Versioning" in SQL Server 2005 Books Online to get started. For a thorough coverage of the Snapshot Isolation options, see Inside SQL Server 2005: The Storage Engine, Chapter 8.)

The snapshot-based isolation levels are based on row versioning, a new feature of the database engine in SQL Server 2005 that serves several purposes. [Row versioning also supports other features such as the inserted and deleted tables in AFTER and INSTEAD OF triggers, online index creating and rebuilding, and MARS (Multiple Active Result Sets), but they are not the focus of this chapter.]

The purpose of the two snapshot-based isolation levels in SQL Server 2005 is to help prevent writers from blocking readers or readers from blocking writers. When you use a snapshot-based isolation level, SQL Server keeps the minimally required number of versions of changed data rows in the version store located in tempdb. Your SELECT statements will read the appropriate prior versions of any data rows that are being changed at the time you try to read them. (SELECT statements inside cursors will also read versioned data.) These versions ensure that your data is consistent and that you only read committed data. Because your statements read the versioned rows, there is no need to take shared locks to ensure that you only read committed data. The removal of shared locks reduces locking and eliminates blocking and deadlocking under the right conditions.

Shared Locking Issues with Snapshot-Based Isolation Levels

There are two important points about SQL Server 2005's snapshot-based isolation levels and shared locking:

- Shared locks will still occur because of automatic lookups, such as a foreign key lookup.
- Sometimes you need to force shared locking, such as with trigger-based referential integrity.

Foreign Key Lookups and Snapshot-Based Isolation Levels

First of all, it's important to realize that not all shared locking is removed by either snapshot-based isolation level. Both options only keep versioned rows for user data and not the database's metadata, so shared locking still occurs on system tables. Such locking on metadata is not normally an issue, and it is beyond our control anyway. But a more important type of shared locking occurs with foreign key lookups on user tables.

Explicit SELECT statements that you issue either under READ COMMITTED SNAPSHOT or in a transaction that is using SNAPSHOT ISOLATION will not take shared locks. But often other DML statements such as INSERT, UPDATE, and DELETE are required to read data in other tables in the process of performing their work. Two examples of this type of indirect reading are foreign key lookups and indexed views. (For brief documentation of this behavior, see SQL Server 2005 Books Online, Understanding Row Version-Based Isolation Levels: Behavior When Modifying Data.)

1. Foreign key lookup under READ COMMITTED. Here's an illustration of how foreign key lookups still require shared locks. Assume you have two tables, a header table called H1 and a detail table called D1, and you have declared a foreign key from D1 to H1 referencing H1's primary key. Here's the code to create that scenario:

```
USE Scratch;
GO
ALTER DATABASE Scratch
    SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE Scratch
    SET READ_COMMITTED_SNAPSHOT OFF;
GO
IF OBJECT_ID('dbo.D1') IS NOT NULL DROP TABLE D1;
IF OBJECT_ID('dbo.H1') IS NOT NULL DROP TABLE H1;
GO
CREATE TABLE H1 (H1ID int NOT NULL PRIMARY KEY, H1Name varchar(10));
CREATE TABLE D1 (D1ID int NOT NULL PRIMARY KEY, H1ID int);
GO
ALTER TABLE D1 ADD CONSTRAINT FK_D1ID_D1
    FOREIGN KEY (H1ID) REFERENCES H1(H1ID);
```

```
GO
INSERT H1 VALUES (1, 'Test');
GO
```

In [Table 6-3](#), Transaction 1 is going to delete the primary key that Transaction 2 is going to look up.

Table 6-3. A Transaction Deleting a Primary Key Referenced by Another Transaction

RC TimeTransaction 1Transaction 21

```
BEGIN TRAN;
DELETE H1
WHERE H1ID = 1;

2

INSERT D1 VALUES (1, 1);
-- Blocked

3COMMIT;-- Insert fails, Error 547
```

This is expected behavior for the default READ COMMITTED isolation level. Transaction 2's INSERT is blocked until it can complete its lookup of the foreign key. After Transaction 1 commits, the lookup fails and the INSERT fails. If you issue the queries against sys.dm_os_waiting_tasks and sys.dm_tran_locks that are shown in the section on blocking earlier in this chapter, you'll see that the DELETE task is holding an exclusive lock on the single row in the H1 table, and the waiting INSERT task is requesting a shared lock. Even though we did not explicitly issue any SELECT statement, the INSERT statement's foreign key lookup took a shared lock.

2. Foreign key lookup under read COMMITTED SNAPSHOT. Now let's change the database to READ COMMITTED SNAPSHOT (sometimes abbreviated to RCSI):

```
ALTER DATABASE Scratch
SET READ_COMMITTED_SNAPSHOT ON;
```

You'll have to make sure for a moment that you only have one connection to the database so that this change will take affect. You also need to reinsert the row into the H1 table.

```
INSERT H1 VALUES (1, 'Test');
```

Now run the code in [Table 6-4](#).

Table 6-4. Similar Transactions in a Database with READ COMMITTED SNAPSHOT Enabled

RCSI TimeTransaction 1Transaction 21

```
BEGIN TRAN;
DELETE H1
WHERE H1ID = 1;

2

INSERT D1 VALUES (1, 1);
-- Blocked

3COMMIT;
```

```
-- Insert fails:  
-- Error 547
```

Again the INSERT will be blocked, showing that in READ COMMITTED SNAPSHOT, foreign key lookups still take shared locks. What is important to notice is that READ COMMITTED SNAPSHOT does not remove shared locks for a foreign key lookup.

As SQL Server Books Online states, the internal lookup of the INSERT statement actually operates under the READ COMMITTED isolation level, not READ COMMITTED SNAPSHOT, so a shared lock is taken by the INSERT statement on the matching row in the H1 table.

3. Foreign key lookup under Full SNAPSHOT isolation. You'll see the same locking behavior when using the SNAPSHOT isolation level, if the transactions are timed correctly. To reproduce this, alter the database to disable RCSI and enable SNAPSHOT isolation (momentarily taking all other connections out of the database):

```
ALTER DATABASE Scratch  
SET READ_COMMITTED_SNAPSHOT OFF;  
ALTER DATABASE Scratch  
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Now reinsert the row into the H1 table one more time.

```
INSERT H1 VALUES (1, 'Test');
```

[Table 6-5](#) shows the same two transactions, but now run the INSERT in a SNAPSHOT isolation transaction.

Table 6-5. Foreign Key Lookup with a Full SNAPSHOT Isolation Transaction

SI Time Transaction 1 Transaction 2

```
BEGIN TRAN;  
DELETE H1  
WHERE H1ID = 1;  
  
2  
  
SET TRANSACTION ISOLATION LEVEL  
SNAPSHOT;  
BEGIN TRAN;  
INSERT D1 VALUES (1, 1);  
-- Blocked  
  
3COMMIT;  
  
-- Insert fails:  
-- Error 3960, update conflict
```

You could make Transaction 1 a SNAPSHOT transaction also, but the behavior is the same. Transaction 2 is still blocked because the INSERT statement internally executes under READ COMMITTED when looking up the foreign key, and is blocked by Transaction 1's uncommitted DELETE. Therefore SNAPSHOT isolation also does not remove shared locks in a foreign key lookup. (When Transaction 1 commits, Transaction 2 produces error 3960 because of an update

conflict. For more on update conflicts, see "Potential SNAPSHOT Isolation Level Conflicts" in the next section of this chapter.)

Other kinds of shared locking that will not be removed by the Snapshot Isolation options include those caused by indexed views referencing more than one table.

Trigger-Based RI and Snapshot Isolation

The Snapshot Isolation options do remove all explicit locking that is caused by SELECT statements that your queries issue. However, this can have an unexpected consequence. For this example, you'll need to re-create the same header and detail tables but this time create a simple trigger that will ensure referential integrity for single-row INSERTs. (Using foreign key constraints is the recommended way to enforce referential integrity, but many databases still use triggers.)

Code View:

```
USE Scratch;
GO
IF OBJECT_ID('dbo.D1') IS NOT NULL DROP TABLE D1;
IF OBJECT_ID('dbo.H1') IS NOT NULL DROP TABLE H1;
CREATE TABLE H1 (H1ID int NOT NULL PRIMARY KEY, H1Name varchar(10));
CREATE TABLE D1 (D1ID int NOT NULL PRIMARY KEY, H1ID int);
GO
CREATE TRIGGER tr_D1_H1
ON D1
FOR INSERT
AS
BEGIN
    -- Single-row inserts only
    IF (SELECT H1ID FROM inserted) NOT IN (SELECT H1ID FROM H1)
        BEGIN
            PRINT 'Rolling back insert'
            ROLLBACK
        END
    END;
END;
GO
INSERT H1 VALUES (1, 'Test')
ALTER DATABASE Scratch SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE Scratch SET READ_COMMITTED_SNAPSHOT OFF;
GO
```

To keep the trigger simple, it just handles single-row inserts. The trigger will produce a run-time error if more than one row is inserted. (I am indebted to Hugo Kornelis's blog on "Snapshot Isolation: A Threat for Integrity?" see the reference in the bibliography, "[Additional Resources and References](#).")

1. Trigger-based RI under READ COMMITTED. In [Table 6-6](#), you can see the same basic steps as shown previously, where the trigger executes and successfully halts the insert of orphaned data.

Table 6-6. Transactions on Tables Using Trigger-Based Referential Integrity in the Default READ COMMITTED Isolation Level

RC Time Transaction 1 Transaction 2

```
BEGIN TRAN;
DELETE H1
```

```

WHERE H1ID = 1;

2

INSERT D1 VALUES (1, 1);
-- Blocked

3COMMIT;-- Insert fails, Error 3609

```

In this case, the trigger in Transaction 2 is blocked because the SELECT statement is running under READ COMMITTED, and Transaction 1 has not finished its COMMIT. When it does, the trigger does not see the row with the primary key in it and rolls back Transaction 2. Before running the next example, reinsert the header row:

```
INSERT H1 VALUES (1, 'Test');
```

2. Trigger-based RI under READ COMMITTED SNAPSHOT. Remove all other users in the database and issue the following commands, putting the database into the READ COMMITTED SNAPSHOT state:

```

ALTER DATABASE Scratch SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE Scratch SET READ_COMMITTED_SNAPSHOT ON;
GO

```

[Table 6-7](#) shows the result of the same steps in RCSI.

Table 6-7. Transactions on Tables Using Trigger-Based Referential Integrity in a Database with READ COMMITTED SNAPSHOT Enabled

RCSI Time Transaction 1 Transaction 2

```

BEGIN TRAN;
DELETE H1
WHERE H1ID = 1;

```

2

```

INSERT D1 VALUES (1, 1);
-- Succeeds!

```

3COMMIT;

Notice what has happened this time: the INSERT succeeds because the trigger's SELECT statement read the version of the primary key row using READ COMMITTED SNAPSHOT. Because no shared lock was issued on the table row being deleted, no blocking occurred. The READ COMMITTED SNAPSHOT option changed the way trigger-based RI works. The SELECT statement in the trigger did not block when it should have, resulting in an orphaned row.

To clean up, empty out the D1 detail table and reinsert the header row into H1:

```

DELETE FROM D1;
INSERT H1 VALUES (1, 'Test');

```

3. Trigger-based RI under the SNAPSHOT isolation level. For this example, remove all other users in the database, set READ COMMITTED SNAPSHOT to OFF in the database, set the database to allow SNAPSHOT isolation:

```

ALTER DATABASE Scratch SET READ_COMMITTED_SNAPSHOT OFF;
ALTER DATABASE Scratch SET ALLOW_SNAPSHOT_ISOLATION ON;

```

```
GO
```

Then issue the following commands in two query windows as shown in [Table 6-8](#).

Table 6-8. Transactions on Tables Using Trigger-Based Referential Integrity in a Database with Full SNAPSHOT Isolation

SI Time Transaction 1 Transaction 2

```
BEGIN TRAN;
DELETE H1
WHERE H1ID = 1;

2

SET TRANSACTION ISOLATION LEVEL
SNAPSHOT;
BEGIN TRAN;
INSERT D1 VALUES (1, 1);
-- Succeeds!!

3COMMIT;COMMIT;
```

The trigger operating inside a SNAPSHOT ISOLATION transaction also did not block, and the INSERT succeeded. The trigger's SELECT statement read the versioned row from the H1 table and did not prevent the row from being inserted. Notice that Transaction 1 in the example runs under READ COMMITTED. You could run it under SNAPSHOT isolation and you'll get the same result, because the isolation level does not affect the exclusive lock on the row being deleted.

What happens with cursors containing SELECT statements inside triggers? The SELECT statements inside cursors behave the same as ordinary SELECT statements whether in a trigger or outside a trigger.

Fixing RI Triggers for Snapshot-Based Isolation Levels

To run referential integrity triggers in databases that might need to have either of the snapshot-based isolation levels set, you need to mimic the behavior of SQL Server and force the SELECT statement in the trigger to issue a shared lock on the primary key row when it checks for foreign keys. For example, you could rewrite the initial RI trigger as follows, using the READCOMMITTEDLOCK hint:

```
CREATE TRIGGER tr_D1_H1
ON D1
FOR INSERT
AS
BEGIN
    -- Single-row inserts only
    IF (SELECT H1ID FROM inserted) NOT IN
        (SELECT H1ID FROM H1 WITH (READCOMMITTEDLOCK))
    BEGIN
        PRINT 'Rolling back insert'
        ROLLBACK
    END
END;
```

The READCOMMITTEDLOCK table hint forces the SELECT statement to use a shared lock when looking up of the primary key value in H1. The statement correctly blocks in the RSCI and SNAPSHOT examples

previously, and then the trigger detects that there is no row after the DELETE commits.

So for triggers, keep two things in mind:

- You need to know whether your database uses triggers for referential integrity lookups, and if so, you need to add a READCOMMITTEDLOCK hint in appropriate places.
- You may also have other triggers that perform other actions based on lookups that must read the actual rows and not versions. In that case, you need to provide READCOMMITTEDLOCK hints to the appropriate SELECT statements.

Note



Trigger failures caused by reading versions rather than locking during lookups is not detectable. You could run consistency checks on your database periodically. But a better solution is to recode your triggers using the READCOMMITTEDLOCK hint and test thoroughly.

Triggers and the OUTPUT Clause

If you use an OUTPUT clause with a trigger, the SELECT statement in the trigger will also revert to locking-based READ COMMITTED. This has the same effect as a READCOMMITTEDLOCK hint on the SELECT statement. The code in [Table 6-9](#) shows an example using the OUTPUT command. The code does not require the trigger with the READCOMMITTEDLOCK hint.

Table 6-9. The OUTPUT Clause with Full SNAPSHOT Isolation and a Referential Integrity Trigger

SI Time Transaction 1 Transaction 2

```
BEGIN TRAN;
DELETE H1
WHERE H1ID = 1;
```

2

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
DECLARE @Result table (InsertedD1ID int,
InsertedH1ID int)
INSERT D1
OUTPUT inserted.D1ID, inserted.H1ID
INTO @Result
VALUES (1, 1)
SELECT * FROM @Result
-- Insert is blocked
```

3 COMMIT; 4 -- Trigger causes a rollback 5

```
COMMIT;
-- Fails because transaction has been rolled
back
```

Transaction 2 generates this message:

```
Rolling back insert
Msg 3609, Level 16, State 1, Line 2
The transaction ended in the trigger. The batch has been aborted.
```

In sum, the OUTPUT clause on a trigger does not read the versioned row of the inserted or deleted table, unlike a SELECT statement within the trigger.

Snapshot Isolation and Triggers

If you use either of the snapshot-based isolation levels with triggers, especially triggers used for referential integrity, you need to be aware that SELECT statements within the trigger will read versioned rows and therefore might cause the trigger to behave in an unexpected way. You can override this by adding the READCOMMITTEDLOCK hint, but then you are adding shared locking back into the transaction, and you may have intended the snapshot-based isolation level to remove shared locking. You need to be aware of the trade-off: You must judge whether the additional shared locking will be sufficiently less than your transactions would have without either of the snapshot-based isolation levels.

Potential SNAPSHOT Isolation Level Conflicts

There are two types of problems that may occur when running transactions under the Full SNAPSHOT isolation level: conflicts with DDL and update conflicts. These two types of conflicts only affect the Full SNAPSHOT isolation level transactions and do not affect transactions running under any of the other isolation levels, including READ COMMITTED SNAPSHOT.

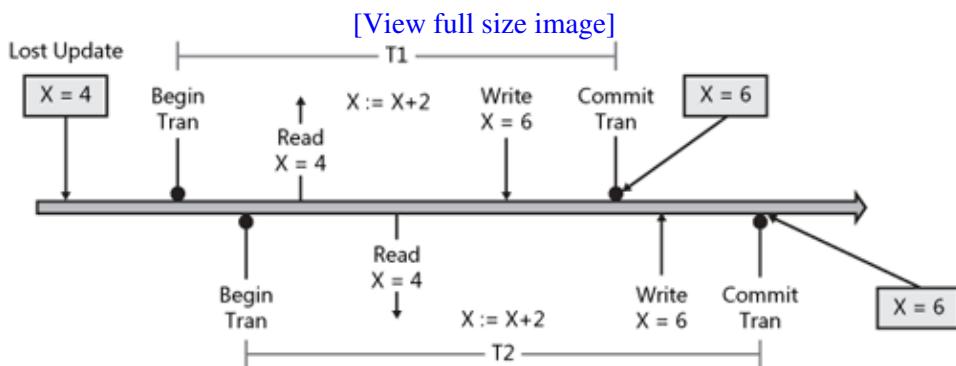
DDL Conflicts

When a Full SNAPSHOT isolation level transaction is running, it is guaranteed a consistent view of the tables it has read until the end of the transaction. After such a transaction reads data from a table, that table may be altered in some fashion that could invalidate the transaction's consistent view of the data. This altering could include added or dropped columns, indexes, constraints, CLR table objects, etc. Any of these changes will cause that particular transaction to fail. (For a more extensive illustration the SNAPSHOT isolation level and DDL conflicts, see Inside SQL Server 2005: The Storage Engine, Chapter 8.)

Update Conflicts: Potential Lost Updates

Update conflicts are only possible with the Full SNAPSHOT isolation level when there is a potential lost update. In SQL Server 2005, an update conflict is an error that aborts a SNAPSHOT transaction in order to prevent a potential lost update. Let's look first at a sample lost update and then view it under the SNAPSHOT and other isolation levels.

In Table 6-5 above, a transaction using Full SNAPSHOT ISOLATION produced an error due to an update conflict. In SQL Server 2005's SNAPSHOT isolation level, update conflicts prevent possible lost updates. To illustrate a lost update, consider a transaction that adds two to a value. Figure 6-7 shows the classic lost update scenario:

Figure 6-7. Illustration of a lost update

The two transactions each read a value from the column X in the same row, add 2 to it, and then write the newly computed value back to the row. When the transactions do not overlap in time, then the number X in the row will be incremented by 4. But if the transactions overlap in time correctly, then one of the updates will overwrite the other, and the value will end up being incremented only by 2. [Table 6-10](#) shows how this scenario uses a row in the AdventureWorks database's Sales.SalesOrderDetail table with the default READ COMMITTED isolation level to illustrate a lost update.

Table 6-10. A Lost Update in the READ COMMITTED Isolation Level

RC Time
Transaction 1
Transaction 2
WAITFOR TIME '<time t>' WAITFOR TIME '<time t>' 2

```
BEGIN TRANSACTION
DECLARE @OrderQty smallint
SET @OrderQty =
(SELECT OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1);
```

```
BEGIN TRANSACTION
DECLARE @OrderQty smallint
SET @OrderQty =
(SELECT OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1);
```

3SET @OrderQty = @OrderQty + 2
SET @OrderQty = @OrderQty + 24

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = @OrderQty
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

5COMMIT;

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = @OrderQty
```

```

WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;

6 COMMIT;

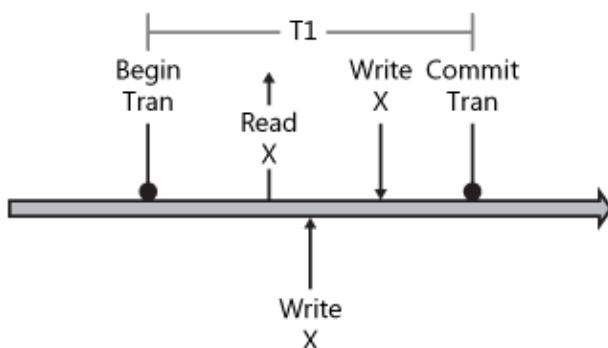
```

Replace the value of <time t> with something a minute or two ahead of the current time, such as WAITFOR TIME '23:30:00', and then execute each transaction in its own Management Studio window. Because each transaction reads the value of OrderQty into a variable, you can use the WAITFOR TIME command to ensure that the transactions run at nearly the same time, so that they will overlap appropriately. The result is that the value will be incremented only by 2, not 4, and so one of the updates is lost. Normally reading data before updating it is not a good programming practice, but it does illustrate the simplest form of a lost update.

When you run these two transactions in the READ COMMITTED isolation level, one of them has its update overwritten, and a lost update occurs. In fact, both the READ UNCOMMITTED and READ COMMITTED isolation levels allow lost updates. That includes the READ COMMITTED SNAPSHOT (RCSI) option as well, because in RCSI, SELECT statements in transactions only read consistent data for the duration of the SELECT statement, and not for the duration of the transaction.

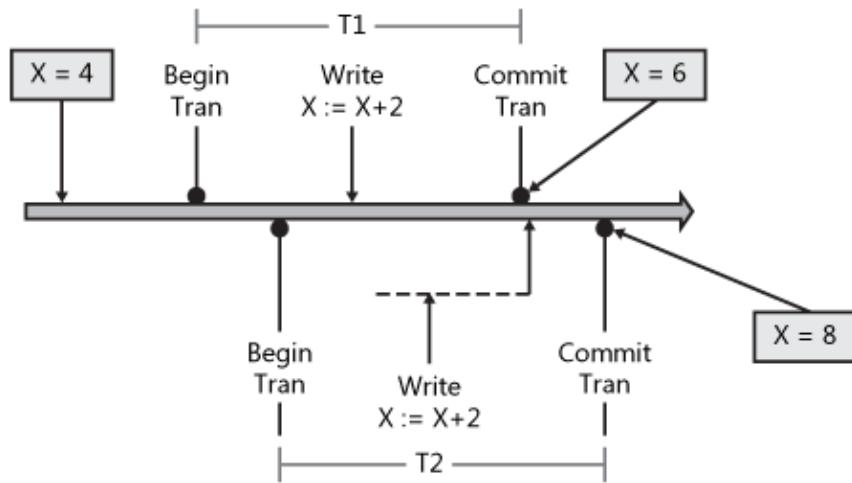
In order to have a lost update, you really only need one transaction to read the data it is about to update. As long as some statement outside the transaction changes the data before the transaction is able to write, there is a potential lost update. Also, the transaction doesn't need to do anything with the data it reads; all it has to do is read it before updating it. [Figure 6-8](#) shows a potential lost update distilled to its simplest form:

Figure 6-8. A distilled potential lost update



The transaction shown above is at least a potential lost update. Because transaction T1 reads the value X before it writes it, and another statement outside the transaction writes it first, T1 is subject to a potential lost update, even if the transaction does nothing with the value after it reads it. The key point about a lost update is that the transaction reads some data first, and another transaction changes the data that was read before the first transaction is able to write it.

Suppose two transactions both write the data but don't read it first at all. What would happen, for example, if you just incremented a row's value in the UPDATE statements and didn't bother to issue the SELECT statement to read the value into a variable? In that case, no lost update would occur, because SQL Server will lock the row for the first UPDATE statement that reaches it, and then will keep it locked to the end of the transaction, and then the next transaction can take over when the first one commits. [Figure 6-9](#) shows two write-only transactions overlapping in time:

Figure 6-9. Write-only transactions never have potential lost updates

Now no matter how you order or overlap the write-only transactions in [Figure 6-7](#), one writer will block the other until it commits, but when the first one finishes, the other can proceed, and both transactions end up succeeding. Since they did not directly or indirectly read the data before their updates, no lost update is possible.

Write-only SNAPSHOT transactions will also not experience potential lost updates because of indirect lookups caused by foreign keys. If one transaction updates the primary key (referenced) table, it will hold an exclusive lock and block the other transaction from referring to it until it is done. If the other transaction updates a referencing table with a foreign key back to the referenced table, even in SNAPSHOT isolation it will take a shared lock on the referenced primary key row and hold it until the transaction completes.

What happens if you take a potential lost update scenario and raise the isolation level to REPEATABLE READ or SERIALIZABLE? In our example from [Figure 6-7](#), a deadlock results. In fact, it's a conversion deadlock: Each transaction is granted a shared lock on the row and keeps the shared lock until the end of the transaction. But each transaction also tries to convert the shared lock into an exclusive lock in order to update the row, and neither can finish. You can use the UPDLOCK hint as one way to resolve this deadlock; see the "[Preventing Lost Updates](#)" section.

Update Conflicts and Full SNAPSHOT Isolation

SQL Server 2005 will not allow even potential lost updates for transactions running in the SNAPSHOT isolation level. To see this, you can run the same example code using the SNAPSHOT isolation level (but without the UPDLOCK hint). The result will be an update conflict. In fact, you can distill the lost update down to a simpler form, as shown in [Table 6-11](#). First, issue the following:

```
ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON;
GO
```

Table 6-11. An Update Conflict Prevents a Potential Lost Update

SI Time Transaction 1 Transaction 2

```
SET TRANSACTION ISOLATION LEVEL
```

SNAPSHOT

```
1WAITFOR TIME '<time t>'WAITFOR TIME '<time t>'2
```

```
BEGIN TRANSACTION
SELECT OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = OrderQty + 2
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

```
3WAITFOR DELAY '00:00:00.2' 4
```

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = OrderQty + 2 WHERE
SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

5

```
-- Transaction aborts with update
conflict error 3960
```

Next, run the following two transactions, each in its own query window. You can use the time settings or run them one step at a time yourself, alternating between windows.

The WAITFOR TIME statements force Transaction 1 to start first, read the row in the Full SNAPSHOT isolation level, and then WAITFOR DELAY for .2 seconds. Transaction 2 just updates the row after Transaction 1 has read it. Transaction 1 will fail. Actually in this case it is only a potential lost update, but SQL Server takes no chances and aborts the transaction with error 3960.

Code View:

```
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'Sales.SalesOrderDetail' directly or in database
'AdventureWorks' to update, delete, or insert the row that has been modified or deleted by
another transaction. Retry the transaction or change the isolation level for the update/
delete statement.
```

Notice the emphasis on the word indirectly that we've placed in bold print in the error message. Can you get an update conflict if you read a table that is indirectly referenced? The answer is yes, as shown in [Table 6-12](#).

Table 6-12. An Update Conflict with an Indirect Reference to a Table**SI TimeTransaction 1Transaction 2**

```
SET TRANSACTION ISOLATION LEVEL
SNAPSHOT
```

1WAITFOR TIME '<time t>'WAITFOR TIME '<time t>'2

```
BEGIN TRANSACTION
SELECT *
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43659;
```

```
UPDATE Sales.SalesOrderHeader
SET RevisionNumber = Revision-
Number + 2
WHERE SalesOrderID = 43659;
```

3WAITFOR DELAY '00:00:00.2' 4

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = OrderQty + 2 WHERE
SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

5

```
-- Transaction aborts with update
conflict error 3960
```

Transaction 1 encounters a potential lost update because it updates a table that indirectly refers to another table that it read earlier in the transaction but was changed before it was able to perform the update. So even indirect references to tables, specifically via foreign keys, can also lead to an update conflict.

Anytime a Full SNAPSHOT isolation level transaction reads some data before it updates the data, the transaction becomes vulnerable to a potential lost update. But if the transaction does not read the data first, no lost update is possible. Look at the following example in [Table 6-13](#).

Table 6-13. Update Conflicts Do Not Occur In Full SNAPSHOT Isolation Level for Write-Only Transactions**SI TimeTransaction 1Transaction 3**

```
SET TRANSACTION ISOLATION LEVEL
SNAPSHOT
```

1WAITFOR TIME '21:43:00.0'WAITFOR TIME '21:43:00.1'2BEGIN TRANSACTION 3

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = OrderQty + 2
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

4WAITFOR DELAY '00:00:00.2' 5

```
UPDATE Sales.SalesOrderDetail
SET OrderQty = OrderQty + 2
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

6COMMIT;

There is no lost update because Transaction 1 did not read the data before trying to write it. Because SQL Server will not execute both UPDATE statements at the same time, there is no lost update and the end result becomes the same no matter how the transactions overlap. But in write-only transactions, there's no advantage to using Full SNAPSHOT isolation over the READ COMMITTED default isolation level. Both isolation levels become equivalent, because write locking behavior does not change across all the isolation levels.

Preventing Lost Updates

As you have seen, the READ COMMITTED and READ UNCOMMITTED isolation levels are vulnerable to potential lost updates. Also as you saw earlier, in the classic lost update scenario, if both transactions run in the REPEATABLE READ or SERIALIZABLE isolation levels they will form a conversion deadlock if the transactions overlap correctly. How can you prevent lost updates?

The most effective method for preventing potential lost updates is to use the UPDLOCK hint. In all the examples so far of lost updates, you can apply the UPDLOCK hint to the SELECT statement that reads the value from Sales.SalesOrderDetail:

```
SET @OrderQty =
(SELECT OrderQty
 FROM Sales.SalesOrderDetail
 WITH (UPDLOCK)
 WHERE SalesOrderID = 43659
 AND SalesOrderDetailID = 1);
```

Because SQL Server only allows one UPDLOCK on a single resource at a time, and the update lock when granted will be held until the transaction ends, the transactions are forced to behave in a serial fashion and no lost update is possible. This technique will also resolve the conversion deadlock if the classic lost update runs in REPEATABLE READ or SERIALIZABLE isolation levels.

Finally, the UPDLOCK hint will also prevent the lost update shown in [Table 6-12](#). Just add the UPDLOCK hint to the SELECT statement in step 2 as follows:

```
SELECT OrderQty
FROM Sales.SalesOrderDetail
WITH (UPDLOCK)
WHERE SalesOrderID = 43659
AND SalesOrderDetailID = 1;
```

You need to be aware of the trade-off here: by adding the UPDLOCK hint you have increased the amount of shared locking and therefore undo the gain you expected from the SNAPSHOT isolation level to begin with. You might be better off by retrying the transaction.

Retrying Transactions to Resolve Update Conflicts

If you have SNAPSHOT isolation level transactions that might be subject to update conflicts, you can prepare for them by adding retry logic to your code. When SQL Server detects a potential lost update and aborts the transaction, the error is trappable using SQL Server 2005's TRY/CATCH constructs. The logic is very similar to that of resolving a deadlock, shown earlier in this chapter. Using the example from the transaction shown in [Table 6-11](#), you could rewrite the SNAPSHOT transaction as:

Code View:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
WAITFOR TIME '00:15:00.0'
DECLARE @Tries tinyint, @Error int
DECLARE @OrderQty smallint
SET @Tries = 1
WHILE @Tries <= 3
BEGIN
    BEGIN TRANSACTION
    BEGIN TRY
        SELECT OrderQty
        FROM Sales.SalesOrderDetail
        WHERE SalesOrderID = 43659
            AND SalesOrderDetailID = 1
        WAITFOR DELAY '00:00:00.2'
        UPDATE Sales.SalesOrderDetail
        SET OrderQty = OrderQty + 2
        WHERE SalesOrderID = 43659
            AND SalesOrderDetailID = 1;
        IF XACT_STATE() = 1 COMMIT
        BREAK
    END TRY
    BEGIN CATCH
        SET @Error = ERROR_NUMBER()
        IF @Error = 3960
        BEGIN
            PRINT '3960 encountered'
            IF XACT_STATE() = -1 ROLLBACK
        END
        SET @Tries = @Tries + 1
        CONTINUE
    END CATCH;
END;
```

The CATCH block traps the error, and the transaction is uncommittable. So the CATCH block rolls it back, and then the CONTINUE command restarts the transaction. If frequency of potential lost updates is low, the restart of the transaction has a reasonable chance of succeeding.

Monitoring the Snapshot Isolation Options

Enabling either of the snapshot isolation database options will cause immediate changes in behavior of an active OLTP database because either option will cause all databases changes to be versioned in tempdb. There will immediately be activity in the tempdb version store, which you may need to troubleshoot, and there may

also be issues related to the way tempdb is configured that you may also need to troubleshoot. Let's take a look at each in turn.

Monitoring Row Versioning

SQL Server 2005 provides you with two main methods for monitoring row versioning: System Monitor (Perfmon) counters and dynamic management views (DMVs for short).

1. Version Store Counters. There are several new SQL Server 2005 counters in the SQLServer: Transactions group of System Monitor. For focusing on the version store, there are three important counters:

- ◆ Version Store Size (KB)
- ◆ Version Generation Rate (KB/Sec)
- ◆ Version Cleanup Rate (KB/Sec)

The Version Store Size (KB) counter shows you much space the version store is using at any particular time. The other two counters show you the rates at which the version store is growing and being cleaned up. In particular, if the Version Generation Rate is persistently greater than the Version Cleanup Rate, you know that tempdb will have to grow in size to support it. If you notice that the Version Cleanup rate is 0, there probably is a long-running transaction preventing the background version cleanup process from doing its work.

The version store uses append-only storage units in tempdb. These are not true tables, and you cannot view them using the usual ways you would inspect objects in a database. For row versioning, SQL Server acquires and frees up space at the unit level. As versioned rows are added and then removed from the version store, SQL Server creates and truncates the version store units. A background thread will remove old storage units once all their versioned rows are no longer needed. The units are not uniformly sized, and a new unit is created roughly once per minute, depending on the level of activity.

Parallel to the version store counters mentioned above, there are three System Monitor counters in the SQLServer: Transactions group that you can use to monitor the unit activity:

- ◆ Version Store unit count
- ◆ Version Store unit creation
- ◆ Version Store unit truncation

These counters all monitor the activity of the version store at the storage unit level. Because the units are not uniform in size, even in a stable system, you might see version store units being created and truncated at different rates. However, if you see the count is increasing, and new units being created, but no units being truncated, that means that some versioned row in an earlier unit is required in the database, most likely by a long-running transaction in the SNAPSHOT isolation level, or perhaps by a long-running SELECT statement for RCSI.

2. Update Conflicts Counter. To monitor update conflicts, you can use the Update conflict ratio in the SQLServer: Transactions group to observe the ratio of failed to successful Full SNAPSHOT isolation level transactions. This counter shows the rate at which Full SNAPSHOT isolation level transactions are encountering potential lost updates and are being aborted due to update conflicts.
3. Page Split Counter. When rows are versioned in a database, a 14-byte pointer is added to each row that is inserted or updated, if they do not already contain it. If you just enable one of the snapshot-based isolation levels on an active database that previously did not have them enabled, you may see page splitting due to the additional pointer. Therefore you should also monitor the Page Splits/sec counter in the SQLServer:Access Methods group.
4. Dynamic Management Views. For a more detailed look at snapshot isolation option activity, you can use a set of dynamic management views. There are quite a number of SQL Server 2005 Dynamic Management Views (DMVs) that you can use to monitor row versioning in general and Full SNAPSHOT isolation level transactions activity in particular. They are grouped with the

sys.dm_tran_* prefix.

For monitoring both of the snapshot-based isolation levels, you can inspect the version store using sys.dm_tran_version_store. The contents of this DMV can be voluminous, but the sys.dm_tran_top_version_generators DMV returns only the top 256 entries, grouped by the database id and the rowset id. For more information on these two DMVs, see their entries in Books Online.

For monitoring Full SNAPSHOT isolation level transactions in particular, you can use the DMV sys.dm_tran_active_snapshot_database_transactions. The following query shows one example:

```
SELECT
    transaction_id,
    session_id,
    transaction_sequence_num,
    is_snapshot,
    max_version_chain_traversed,
    elapsed_time_seconds
FROM sys.dm_tran_active_snapshot_database_transactions
ORDER BY elapsed_time_seconds DESC;
```

The session_id will tell you which session is using a Full SNAPSHOT isolation level transaction, and elapsed_time_seconds will tell you how long the transaction has been open.

Take a close look at the max_version_chain_traversed column. It shows the size of the longest version chain that a given transaction must traverse to find its appropriate versioned row. When a Full SNAPSHOT isolation transaction reads any data row, from that point in time on, if the row is modified by other shorter transactions, multiple versions of the row must be kept in the version store and linked together. Other Full SNAPSHOT isolation transactions will have to traverse that version chain to find their appropriate row in the list. If your Full SNAPSHOT isolation transactions are short, and the data they read is not changed often outside those transactions, the version chain should be small. (For more information about version chains, see Inside SQL Server 2005: The Storage Engine, Chapter 8.)

Inside a transaction, you can find out your own transaction row versioning information by using sys.dm_tran_current_transaction. If the transaction is running under Full SNAPSHOT isolation, the transaction_is_snapshot column will be 1.

You can also query sys.dm_tran_transactions_snapshot to find out how many Full SNAPSHOT isolation transactions are currently active. The additional columns will show what other SNAPSHOT transactions have made modifications to rows for which a given transaction must read versions.

Monitoring

In SQL Server 2005, the behavior of the tempdb database gains in importance because many new operations require it. The most prominent are those requiring use of the version store.

For the Snapshot Isolation options, the most important tempdb issues concern the size and behavior of the version store. You need to make sure that tempdb has sufficient free space and I/O throughput to handle the version store load in addition to other activities that occur in tempdb caused by queries, temporary tables, and table variables, as well as work tables for queries. The version store is also used for online index rebuilds, as well as triggers (for constructing the inserted and deleted virtual tables), and MARS operations.

You can monitor the free space in tempdb using the free space in tempdb counter in System Monitor (Perfmon). You can also inspect the free space in tempdb using sys.dm_db_file_space_usage. The following query shows one way of using this DMV to inspect the free space in tempdb:

```

SELECT
    SUM(user_object_reserved_page_count) * 8.192 AS UserObjectsKB,
    SUM(internal_object_reserved_page_count) * 8.192 AS InternalObjectsKB,
    SUM(version_store_reserved_page_count) * 8.192 AS VersionStoreKB,
    SUM(unallocated_extent_page_count) * 8.192 AS FreeSpaceKB
FROM sys.dm_db_file_space_usage;

```

In SQL Server 2005, sys.dm_db_file_space_usage only reports information about the tempdb database. You can run it from any database context, but it only reports tempdb information. (The above query was adapted from the Microsoft white paper, "Working with tempdb in SQL Server 2005" see the reference in the bibliography, "[Additional Resources and References](#).")

You need to ensure that tempdb has enough free space to operate the row versioning store because if tempdb becomes full, new row version unit store creation will stop. Other database activity will not be affected, but then new object creation in tempdb may halt. When row versions generation stops, Full SNAPSHOT isolation transactions, as well as other operations requiring row versioning, will fail if they cannot access the correct versioned row. For more information, see SQL Server Books Online, Row Versioning Resource Usage.

When using either of the snapshot-based isolation levels, you should take care to get a baseline estimate of tempdb activity without the options enabled. When you enable either snapshot-based isolation level, you can then observe their impact on tempdb behavior.

Resolving Snapshot Isolation Problems

When you enable either the READ COMMITTED SNAPSHOT or the Full SNAPSHOT isolation level options, there are a number of techniques you can use to resolve issues in the row versioning store, tempdb, and in the Snapshot Isolation options.

Resolving Row Versioning Issues

Row versioning problems can be observed by using the techniques described previously by monitoring. Problems with resolutions include:

- Enabling row versioning for either Snapshot Isolation option is blocked or causing transactions to fail. Changing the database state to row versioning for either of the snapshot-based isolation levels has certain requirements. Make sure that you enable either row versioning options when the database is offline or at least during low usage times.
- Page splitting after converting to row versioning. You should observe the amount of page splitting on a database that has been recently changed to snapshot isolation. Page splits may occur due to the additional 14-byte pointers that will be added to rows. You may need to rebuild clustered indexes on tables if the level of fragmentation rises.
- Unrestricted growth of the version store. Use the System Monitor (Perfmon) counters to make sure the version cleanup rate keeps up with the version creation rate. If you observe that the row versioning store is growing and not cleaning up properly, it is probably due to long-running transactions. You can identify the transactions by querying sys.dm_tran_active_snapshot_database_transactions, and then by inspecting the code for the transactions.

Resolving Issues Caused by Snapshot Isolation

The version store is located in tempdb and enabling either of the snapshot-based isolation levels will immediately cause additional activity in tempdb.

- Running out of free space in tempdb. If you find that the free space in tempdb is declining, you may need to size it larger to handle the appropriate load.
- Online indexing operations interfering with SI and RCSI. Online indexing also uses the version store, though a different part of it than the storage units used by the snapshot-based isolation levels. If online indexing causes too large a growth of the version store and therefore a loss of free space in tempdb, you may need to schedule online indexing operations for low-usage periods so as not to interfere with the snapshot isolation version store activity.
- Tempdb I/O issues. If you observe disk queuing on tempdb, you need to increase the disk throughput. It is generally recommended that you create multiple data files for tempdb, at least one file for each CPU on the server, because SQL Server can make use of asynchronous I/O when writing to tempdb. Another option is to split the tempdb workload among multiple instances of SQL Server, as each SQL Server instance will have its own tempdb database.

Resolving READ COMMITTED SNAPSHOT Issues

The READ COMMITTED SNAPSHOT option will ensure that explicit SELECT statements, whether in queries or in triggers, will read versioned rows and not require shared locks on data that is undergoing change. Each individual SELECT statement will return information about committed data as of the beginning of the statement. The following issues may arise:

- Enabling the READ COMMITTED SNAPSHOT option in a database is blocked. If there are any users in the database other than the session that is enabling the database, the enabling command will be blocked until those users are removed. Therefore you need to enable the RCSI option during database downtime, and you can temporarily remove all other users.
- Orphan rows are appearing in a database that uses trigger-based referential integrity. SELECT statements in triggers that enforce referential integrity will read the version rows of deleted primary key rows, and therefore may fail. This behavior is not an error, and there is no good method for detecting it other than querying the tables. The best solution is to replace the trigger-based referential integrity with declared foreign keys. If that is not feasible, you can at least add the READCOMMITTEDLOCK hint to the triggers and override reading the row versions. However, this will cause additional shared locks to occur and increase the chance of blocking and potentially deadlocking.

Resolving SNAPSHOT Isolation Level Issues

The Full SNAPSHOT isolation level ensures that all SELECT statements in a SNAPSHOT isolation level transaction will return data that is committed as of the first data access operation of the transaction. This is a stronger level of consistency than READ COMMITTED SNAPSHOT, and brings with it special problems. The following issues may arise when using the SNAPSHOT isolation level:

- The database transition to Full SNAPSHOT isolation level does not complete. If you enable SNAPSHOT isolation in a database that has currently running non-SNAPSHOT isolation level transactions, the database will remain in a IN_TRANSITION_TO_ON state, which you can observe in the sys.databases catalog view. If that state continues for an unacceptable length of time, you can either kill the blocking transaction or cancel the attempt to enable it and try later.
- Transactions are failing because of update conflicts. If you observe a high rate of update conflicts using the System Monitor Update Conflict ratio counter, you need to inspect the transaction using SNAPSHOT isolation. You need to identify the transactions containing SELECT statements that are reading data that has been subsequently changed before the transaction has its ability to update the data. In some cases you may be able to remove the SELECT and make the transaction write-only or write-mostly, avoiding reads of data that must be updated. You may also be able to change the transaction back to READ COMMITTED. If neither of these are possible, you can add a HOLDLOCK hint to the SELECT statement to keep the shared locks for the duration of the

transaction. A better solution may be to retry the transactions by using the TRY/CATCH features of Transact-SQL.

- Transactions are failing due to DDL conflicts. The SNAPSHOT isolation level requires that the underlying tables that have been read by SELECT statements not be altered. The best solution here is to make sure that DDL operations occur during a time when no affected SNAPSHOT isolation transactions are running.
- SELECT queries within SNAPSHOT transactions are running much slower. This may be because they are traversing long row versioning chains. You can detect this by querying the max_version_chain_traversed column in sys.dm_tran_active_snapshot_database_transactions. Long version chains, more than a few deep, indicate that the data being read by SNAPSHOT isolation transactions is being changed often. This could be an indication that your transactions are too long for the type of work they do. It could also be an indication that the database update activity is simply too high to support SNAPSHOT isolation.
- Transactions are failing when referencing global temporary tables. When SNAPSHOT isolation level transactions reference global temporary tables, you must either enable the SNAPSHOT isolation level in tempdb or add a READCOMMITTED lock when referencing those tables. See "Using Row Versioning-based Isolation Levels" in SQL Server 2005 Books Online.

Appropriate Uses for Snapshot Isolation

The SQL Server 2005 snapshot-based isolation levels can help reduce shared locking, and thereby blocking and deadlocking in a database. Not all shared locking is removed: shared locks are still taken for foreign key lookups and some indexed view operations when multiple tables are involved.

You should make your decision about using either of the snapshot-based isolation levels by weighing the costs and benefits of each. For a discussion of these costs and benefits, see Choosing Row Versioning-based Isolation Levels in SQL Server 2005 Books Online.

Use Cases for READ COMMITTED SNAPSHOT

Enabling the READ COMMITTED SNAPSHOT database causes single SELECT statements to read row versions instead of placing shared locks on data. In addition, the SELECT statement will return a view of its data that is consistent and committed at the point in time that the SELECT statement started. Therefore the major uses of the READ COMMITTED SNAPSHOT option are:

- Reducing shared locks, including lock escalation of shared locks.
- Reducing or eliminating reader/writer blocking and deadlocking.
- Providing a consistent view of committed data during a single SELECT statement without locking.

One example use of READ COMMITTED SNAPSHOT is an OLTP database that has a relatively low rate of update activity and high rate of read activity. Assume that the database uses declarative referential integrity and that the update and read queries are otherwise well-tuned. If read queries are being commonly blocked by occasional writers for unacceptably long times, then provided the tempdb database can handle the row versioning load, this database would benefit from READ COMMITTED SNAPSHOT.

Another good use case is that of queries against a reporting server that is a transactional replication subscriber. Report queries may be blocked by updates sent from the replication publisher, or they may be deadlock victims when conflicting with those updates. If you enable READ COMMITTED SNAPSHOT on the subscriber database, the updates from replication will cause activity in the version store of tempdb. Often that is not a problem because the subscriber only has a subset of the publisher's data. Even if the publisher database uses triggers extensively, or has trigger-based referential integrity, triggers are not normally replicated to a subscriber because replication sends all the relevant changes from the transaction log. Once READ COMMITTED SNAPSHOT is enabled, the SELECT statements in the reporting queries will no longer

take shared locks and the reports will not longer be blocked by, or deadlock with, replication.

Use Cases for Full SNAPSHOT Isolation

One good use case for the SNAPSHOT isolation level is a data warehouse or operational data store that has low update activity and periodic inserts of new data. Inserts are not normally versioned, so the periodic loading of new data will likely not cause activity in the version store. Assume the queries require multiple SELECT statements that deliver results and aggregations that must be consistent with each other. In this case, running those read queries in a SNAPSHOT transaction will provide consistent results and not interfere with the periodic loading, because shared locking on the data rows will be reduced or even removed. Read-only transactions can be an excellent use of the SNAPSHOT isolation level.

However, once you introduce write activity into a SNAPSHOT isolation transaction, you must take care to avoid a number of issues: update conflicts, DDL conflicts, orphaned data in databases that use triggers extensively including for referential integrity, and so on. In addition, writer/writer blocking and deadlocking is not prevented by the SNAPSHOT isolation level, so you must be alert for those possibilities as well.

Summary

Troubleshooting concurrency issues in SQL Server 2005 primarily involves dealing with and resolving locking, blocking, and deadlocking issues. SQL Server 2005 provides you many new tools for detecting, analyzing, and resolving these issues. For detecting and resolving blocking, new dynamic management views allow you to monitor waiting tasks and discover the queries causing blocking. In addition, the new SQL Trace Blocked Process Report event class allows you to view blocking meeting a certain threshold and store the results for later analysis. For detecting and resolving deadlocks, SQL Server 2005 provides new deadlock reporting options with the 1222 trace flag and the SQL Trace Deadlock Graph event class. For reducing shared locking, and thereby reducing or eliminating reader/writer blocking and deadlocking, you can use the new snapshot-based isolation levels. However, it is important to weigh the benefits of the snapshot-based isolation levels against the costs of additional activity in tempdb and potential issues such as update conflicts.

Additional Resources and References

The following papers and blogs from the SQL Server engineering team can help you find additional information and ideas about performance debugging.

Agarwal, Sunil; Boris Baryshnikov; Tom Davidson; Keith Elmore; Denzil Ribeiro; and Juergen Thomas. "Troubleshooting Performance Problems in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx>, October 2005.

Marathe, Aurun. "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx>, July 2004.

Patel, Burzin A. "Forcing Query Plans." <http://www.microsoft.com/technet/prodtechnol/sql/2005/frcqupln.mspx>, November 2005.

"SQL Server Engine Tips." <http://blogs.msdn.com/sqltips>. SQL Server Engine Dev blog.

"Tips, Tricks, and Advice from the SQL Server Query Optimization Team." <http://blogs.msdn.com/queryoptteam>. SQL Server QO Team blog.

The following papers provide more detail about some of the SQL Server query processing features.

Blakeley, José A.; Conor Cunningham; Nigel Ellis; Balaji Rathakrishnan; and Ming-Chuan Wu. "Distributed/Heterogeneous Query Processing in Microsoft SQL Server." ICDE 2005: 1001–1012.

Galindo-Legaria, César A.; Stefano Stefani; and Florian Waas. "Query Processing for SQL Updates." SIGMOD Conference 2004: 844–849.

Hanson, Eric N. "Improving Performance with SQL Server 2005 Indexed Views." <http://www.microsoft.com/technet/prodtechnol/sql/2005/ipsql05iv.mspx>, March 2005.

Hanson, Eric N. "Statistics Used by the Query Optimizer in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx>, April 2005.

Kleinerman, Christian. "Multiple Active Result Sets (MARS) in SQL Server 2005." <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/MARSinSQL05.asp>, June 2005.

The following papers provide more detail about the principles behind query optimization and execution in SQL Server.

Galindo-Legaria, César A.; and Milind Joshi. "Orthogonal Optimization of Subqueries and Aggregation." SIGMOD Conference 2001: 571–581.

Graefe, Goetz. "Query Evaluation Techniques for Large Databases." ACM Computing Surveys, 25(2): 73–170 (1993).

Pellenkof, Arjan; César A. Galindo-Legaria; and Martin L. Kersten. "The Complexity of Transformation-Based Join Enumeration." VLDB 1997: 306–315.

Waas, Florian; and César A. Galindo-Legaria. "Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer." SIGMOD Conference 2000: 499–509.

The following book chapter provides more detail about locking and concurrency.

Delaney, Kalen. Inside SQL Server 2005: The Storage Engine. Microsoft Press, 2007. See Chapter 8, "Locking and Concurrency." This chapter thoroughly covers the internals and behavior of locking, lock compatibility, and the Snapshot Isolation options.

The following book chapters, blogs, articles, and white papers provide more information about locking, blocking, and deadlocking.

Ben-Gan, Itzik. Inside Microsoft SQL Server 2005: T-SQL Programming. Microsoft Press, 2006. See Chapter 10, "Exception Handling."

Dorr, Bob. "How to Diagnose and Correct Errors 17883, 17884, 17887, and 17888." TechNet white paper at <http://www.microsoft.com/technet/prodtechnol/sql/2005/diagandcorrecterrs.mspx>.

Duffy, Joe. "Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps." <http://msdn.microsoft.com/msdnmag/issues/06/04/Deadlocks/default.aspx>.

Duncan, Bart. "Deadlock Troubleshooting." <http://blogs.msdn.com/bartd/default.aspx>. MSDN blog in three parts. Nice coverage of the 1222 trace flag, deadlocks caused by index access, and recommends use of the Database Tuning Advisor for suggesting new indexes.

Henderson, Ken, ed. SQL Server 2005 Practical Troubleshooting: The Database Engine. Addison-Wesley, 2007. See Chapter 1, "Waiting and Blocking Issues," by Santeri Voutilainen. Contains a thorough discussion of wait types and a discussion of blocking due to latching.

Kollar, Lubor. "Previously Committed Rows Might Be Missed If NOLOCK Hint Is Used." <http://blogs.msdn.com/sqlcat/archive/2007/02/01/Previously-Committed-Rows-Might-Be-Missed-if-NoLock-Hint-is-Used.aspx>. This blog points out some potential issues with the NOLOCK hint and provides sample code to reproduce the issue.

The following references provide more information about row versioning in SQL Server.

Agarwal, Sunil; Boris Baryshnikov; Tom Davidson; Keith Elmore; Denzil Ribeiro; and Juergen Thomas . "Troubleshooting Performance Problems in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx#EUVAE>. Contains a good discussion of potential causes of contention in tempdb.

Kornelis, Hugo. "Snapshot Isolation: A Threat for Integrity?" http://sqlblog.com/blogs/hugo_kornelis/archive/2006/07/21/Snapshot_and_integrity_part_1.aspx. In four parts, this blog brings out some potential issues when row versioning is used with triggers.

SQL Server 2005 Books Online, "Row Versioning [SQL Server]." Essential introduction to row versioning and the Snapshot Isolation options. This group of topics is well organized and full of important details and clear explanations.

Tripp, Kimberly; and Neal Graves. "SQL Server 2005 Row Versioning-Based Transaction Isolation." <http://msdn2.microsoft.com/en-us/library/ms345124.aspx>. Contains an extensive discussion of use cases for the Snapshot Isolation options.

Xiao, Wei; Matt Hink; Mirek Sztajno; and Sunil Agarwal. "Working with tempdb in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx>. Contains important information and advice about tempdb behavior and management.

Additional Resources and References

The following papers and blogs from the SQL Server engineering team can help you find additional information and ideas about performance debugging.

Agarwal, Sunil; Boris Baryshnikov; Tom Davidson; Keith Elmore; Denzil Ribeiro; and Juergen Thomas. "Troubleshooting Performance Problems in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx>, October 2005.

Marathe, Aurun. "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx>, July 2004.

Patel, Burzin A. "Forcing Query Plans." <http://www.microsoft.com/technet/prodtechnol/sql/2005/frcqupln.mspx>, November 2005.

"SQL Server Engine Tips." <http://blogs.msdn.com/sqltips>. SQL Server Engine Dev blog.

"Tips, Tricks, and Advice from the SQL Server Query Optimization Team." <http://blogs.msdn.com/queryoptteam>. SQL Server QO Team blog.

The following papers provide more detail about some of the SQL Server query processing features.

Blakeley, José A.; Conor Cunningham; Nigel Ellis; Balaji Rathakrishnan; and Ming-Chuan Wu. "Distributed/Heterogeneous Query Processing in Microsoft SQL Server." ICDE 2005: 1001–1012.

Galindo-Legaria, César A.; Stefano Stefani; and Florian Waas. "Query Processing for SQL Updates." SIGMOD Conference 2004: 844–849.

Hanson, Eric N. "Improving Performance with SQL Server 2005 Indexed Views." <http://www.microsoft.com/technet/prodtechnol/sql/2005/ipsql05iv.mspx>, March 2005.

Hanson, Eric N. "Statistics Used by the Query Optimizer in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx>, April 2005.

Kleinerman, Christian. "Multiple Active Result Sets (MARS) in SQL Server 2005." <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/MARSinSQL05.asp>, June 2005.

The following papers provide more detail about the principles behind query optimization and execution in SQL Server.

Galindo-Legaria, César A.; and Milind Joshi. "Orthogonal Optimization of Subqueries and Aggregation." SIGMOD Conference 2001: 571â 581.

Graefe, Goetz. "Query Evaluation Techniques for Large Databases." ACM Computing Surveys, 25(2): 73â 170 (1993).

Pellenkoft, Arjan; César A. Galindo-Legaria; and Martin L. Kersten. "The Complexity of Transformation-Based Join Enumeration." VLDB 1997: 306â 315.

Waas, Forian; and César A. Galindo-Legaria. "Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer." SIGMOD Conference 2000: 499â 509.

The following book chapter provides more detail about locking and concurrency.

Delaney, Kalen. Inside SQL Server 2005: The Storage Engine. Microsoft Press, 2007. See Chapter 8, "Locking and Concurrency." This chapter thoroughly covers the internals and behavior of locking, lock compatibility, and the Snapshot Isolation options.

The following book chapters, blogs, articles, and white papers provide more information about locking, blocking, and deadlocking.

Ben-Gan, Itzik. Inside Microsoft SQL Server 2005: T-SQL Programming. Microsoft Press, 2006. See Chapter 10, "Exception Handling."

Dorr, Bob. "How to Diagnose and Correct Errors 17883, 17884, 17887, and 17888." TechNet white paper at <http://www.microsoft.com/technet/prodtechnol/sql/2005/diagandcorrecterrs.mspx>.

Duffy, Joe. "Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps." <http://msdn.microsoft.com/msdnmag/issues/06/04/Deadlocks/default.aspx>.

Duncan, Bart. "Deadlock Troubleshooting." <http://blogs.msdn.com/bartd/default.aspx>. MSDN blog in three

parts. Nice coverage of the 1222 trace flag, deadlocks caused by index access, and recommends use of the Database Tuning Advisor for suggesting new indexes.

Henderson, Ken, ed. *SQL Server 2005 Practical Troubleshooting: The Database Engine*. Addison-Wesley, 2007. See Chapter 1, "Waiting and Blocking Issues," by Santeri Voutilainen. Contains a thorough discussion of wait types and a discussion of blocking due to latching.

Kollar, Lubor. "Previously Committed Rows Might Be Missed If NOLOCK Hint Is Used." <http://blogs.msdn.com/sqlcat/archive/2007/02/01/Previously-Committed-Rows-Might-Be-Missed-if-NoLock-Hint-is-Used.aspx>. This blog points out some potential issues with the NOLOCK hint and provides sample code to reproduce the issue.

The following references provide more information about row versioning in SQL Server.

Agarwal, Sunil; Boris Baryshnikov; Tom Davidson; Keith Elmore; Denzil Ribeiro; and Juergen Thomas . "Troubleshooting Performance Problems in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx#EUVAE>. Contains a good discussion of potential causes of contention in tempdb.

Kornelis, Hugo. "Snapshot Isolation: A Threat for Integrity?" http://sqlblog.com/blogs/hugo_kornelis/archive/2006/07/21/Snapshot_and_integrity_part_1.aspx. In four parts, this blog brings out some potential issues when row versioning is used with triggers.

SQL Server 2005 Books Online, "Row Versioning [SQL Server]." Essential introduction to row versioning and the Snapshot Isolation options. This group of topics is well organized and full of important details and clear explanations.

Tripp, Kimberly; and Neal Graves. "SQL Server 2005 Row Versioning-Based Transaction Isolation." <http://msdn2.microsoft.com/en-us/library/ms345124.aspx>. Contains an extensive discussion of use cases for the Snapshot Isolation options.

Xiao, Wei; Matt Hink; Mirek Sztajno; and Sunil Agarwal. "Working with tempdb in SQL Server 2005." <http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx>. Contains important information and advice about tempdb behavior and management.

About the Authors

[Kalen Delaney](#)

[Sunil Agarwal](#)

[Craig Freedman](#)

[Adam Machanic](#)

Ron Talmage

Kalen Delaney



Kalen Delaney earned an M.S. in Computer Science from U.C. Berkeley and then worked as a university and college Computer Science instructor for eight years. Kalen has been working with SQL Server since 1987 when she joined the Sybase Corporation in Berkeley, California, where she worked in both the technical support department and the training organization. Kalen has been an independent trainer and consultant since 1992. She has worked with both the Microsoft and Sybase Corporations to develop courses and provide internal training for their technical support staff. Kalen has taught Microsoft Official Curriculum (MOC) courses, as well as her own independently developed courses to clients around the world. In addition, she has been writing regularly about SQL Server since 1995. Kalen is also a contributing editor and columnist for SQL Server Magazine and has been a SQL Server MVP since 1993. You can reach Kalen through her Web site at <http://www.InsideSQLServer.com>.

About the Authors

[Kalen Delaney](#)

[Sunil Agarwal](#)

[Craig Freedman](#)

[Adam Machanic](#)

[Ron Talmage](#)

Kalen Delaney



Kalen Delaney earned an M.S. in Computer Science from U.C. Berkeley and then worked as a university and college Computer Science instructor for eight years. Kalen has been working with SQL Server since 1987 when she joined the Sybase Corporation in Berkeley, California, where she worked in both the technical support department and the training organization. Kalen has been an independent trainer and consultant since 1992. She has worked with both the Microsoft and Sybase Corporations to develop courses and provide

internal training for their technical support staff. Kalen has taught Microsoft Official Curriculum (MOC) courses, as well as her own independently developed courses to clients around the world. In addition, she has been writing regularly about SQL Server since 1995. Kalen is also a contributing editor and columnist for SQL Server Magazine and has been a SQL Server MVP since 1993. You can reach Kalen through her Web site at <http://www.InsideSQLServer.com>.

Sunil Agarwal



Sunil Agarwal is a Senior Program Manager in SQL Server Storage Engine Group. Prior to joining Microsoft, Sunil had worked at Digital Equipment Corporation, Sybase, BMC Software, and DigitalThink with primary focus on core database engine technology and related applications. Sunil holds a B.S. in Electrical Engineering from Indian Institute of Technology, Kanpur, India, and has done graduate work in Computer Science at University of Rhode Island and at Brown University.

Craig Freedman



Craig Freedman has more than a decade of experience designing and implementing relational database servers, including six years as a Software Design Engineer with the Microsoft SQL Server query processing team. He holds a Ph.D. from the University of Wisconsinâ Madison where he researched parallel file systems and video on demand. Craig enjoys living and playing in the Pacific Northwest with his wife and two young children.

Adam Machanic



Adam Machanic is an independent database software consultant, writer, and speaker based in Boston, Massachusetts. He has implemented SQL Server solutions for a variety of high-availability OLTP and

large-scale data warehouse applications, and also specializes in .NET data access layer performance optimization. Adam has written for CoDe, TechNet, and VSJ magazines, and has contributed to several books on SQL Server, including Expert SQL Server 2005 Development (Apress, 2007). He regularly speaks at user groups, community events, and conferences on a variety of SQL Server and .NET-related topics. He is a Microsoft Most Valuable Professional (MVP) for SQL Server and a Microsoft Certified IT Professional (MCITP).

When not sitting at the keyboard pounding out code or code-related prose, Adam tries to spend a bit of time with his wife, Kate, and daughter, Aura, both of whom seem to believe that there is more to life than SQL.

Adam blogs at <http://www.sqlblog.com>, and can be contacted directly at amachanic@datamanipulation.net.

Ron Talmage



Ron Talmage is a co-founder and Practice Manager with Solid Quality Learning. He has over twenty years of experience in the IT world, and has been providing mentoring, teaching, and conference presentations on SQL Server since version 4.21. He is a SQL Server MVP and current president of the Pacific Northwest SQL Server Users Group. He writes regularly for CoDe Magazine online and SQL Server Magazine. You can reach him at Ron@SolidQualityLearning.com.

Additional Resources for Developers: Advanced Topics and Best Practices

Published and Forthcoming Titles from Microsoft Press

Code Complete, Second Edition
Steve McConnell â € ISBN 0-7356-1967-0

For more than a decade, Steve McConnell, one of the premier authors and voices in the software community, has helped change the way developers write codeâ and produce better software. Now his classic book, Code Complete, has been fully updated and revised with best practices in the art and science of constructing software. Topics include design, applying good techniques to construction, eliminating errors, planning, managing construction activities, and relating personal character to superior software. This new edition features fully updated information on programming techniques, including the emergence of Web-style programming, and integrated coverage of object-oriented design. You'll also find new code examplesâ both good and badâ in C++, Microsoft® Visual Basic®, C#, and Java, although the focus is squarely on techniques and practices.

More About Software Requirements: Thorny Issues and Practical Advice
Karl E. Wiegers â € ISBN 0-7356-2267-1

Have you ever delivered software that satisfied all of the project specifications, but failed to meet any of the customers expectations? Without formal, verifiable requirementsâ and a system for managing themâ the result is often a gap between what developers think they're supposed to build and what customers think they're

going to get. Too often, lessons about software requirements engineering processes are formal or academic, and not of value to real-world, professional development teams. In this follow-up guide to Software Requirements, Second Edition, you will discover even more practical techniques for gathering and managing software requirements that help you deliver software that meets project and customer specifications. Succinct and immediately useful, this book is a must-have for developers and architects.



Software Estimation: Demystifying the Black Art

Steve McConnell â € ISBN 0-7356-0535-1

Often referred to as the "black art" because of its complexity and uncertainty, software estimation is not as hard or mysterious as people think. However, the art of how to create effective cost and schedule estimates has not been very well publicized. Software Estimation provides a proven set of procedures and heuristics that software developers, technical leads, and project managers can apply to their projects. Instead of arcane treatises and rigid modeling techniques, award-winning author Steve McConnell gives practical guidance to help organizations achieve basic estimation proficiency and lay the groundwork to continue improving project cost estimates. This book does not avoid the more complex mathematical estimation approaches, but the non-mathematical reader will find plenty of useful guidelines without getting bogged down in complex formulas.

Debugging, Tuning, and Testing Microsoft .NET 2.0 Applications

John Robbins â € ISBN 0-7356-2202-7

Making an application the best it can be has long been a time-consuming task best accomplished with specialized and costly tools. With Microsoft Visual Studio® 2005, developers have available a new range of built-in functionality that enables them to debug their code quickly and efficiently, tune it to optimum performance, and test applications to ensure compatibility and trouble-free operation. In this accessible and hands-on book, debugging expert John Robbins shows developers how to use the tools and functions in Visual Studio to their full advantage to ensure high-quality applications.

The Security Development Lifecycle

Michael Howard and Steve Lipner â € ISBN 0-7356-2214-0

Adapted from Microsoft's standard development process, the Security Development Lifecycle (SDL) is a methodology that helps reduce the number of security defects in code at every stage of the development process, from design to release. This book details each stage of the SDL methodology and discusses its implementation across a range of Microsoft software, including Microsoft Windows Serverâ € 2003, Microsoft SQL Serverâ € 2000 Service Pack 3, and Microsoft Exchange Server 2003 Service Pack 1, to help measurably improve security features. You get direct access to insights from Microsoft's security team and lessons that are applicable to software development processes worldwide, whether on a small-scale or a large-scale. This book includes a CD featuring videos of developer training classes.

Software Requirements, Second Edition

Karl E. Wiegers â € ISBN 0-7356-1879-8

Writing Secure Code, Second Edition

Michael Howard and David LeBlanc â € ISBN 0-7356-1722-8

CLR via C#, Second Edition

Jeffrey Richter â € ISBN 0-7356-2163-2

For more information about Microsoft Press® books and other learning products, visit:

www.microsoft.com/mspress and www.microsoft.com/learning



Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at www.microsoft.com/mspress. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call 1-800-268-2222.)

Additional SQL Server Resources for Developers

Published and Forthcoming Titles from Microsoft Press

Microsoft® SQL Server® 2005 Express Edition Step by Step

Jackie Goldstein â € ISBN 0-7356-2184-5

Teach yourself how to get database projects up and running quickly with SQL Server Express Editionâ – a free, easy-to-use database product that is based on SQL Server 2005 technology. It's designed for building simple, dynamic applications, with all the rich functionality of the SQL Server database engine and using the same data access APIs, such as Microsoft ADO.NET, SQL Native Client, and T-SQL. Whether you're new to database programming or new to SQL Server, you'll learn how, when, and why to use specific features of this simple but powerful database development environment. Each chapter puts you to work, building your knowledge of core capabilities and guiding you as you create actual components and working applications.



Microsoft SQL Server 2005 Programming Step by Step

Fernando Guerrero â € ISBN 0-7356-2207-8

SQL Server 2005 is Microsoft's next-generation data management and analysis solution that delivers enhanced scalability, availability, and security features to enterprise data and analytical applications while making them easier to create, deploy, and manage. Now you can teach yourself how to design, build, test, deploy, and maintain SQL Server databasesâ – one step at a time. Instead of merely focusing on describing new features, this book shows new database programmers and administrators how to use specific features within typical business scenarios. Each chapter provides a highly practical learning experience that demonstrates how to build database solutions to solve common business problems.



Microsoft SQL Server 2005 Analysis Services Step by Step
 Hitachi Consulting Services ª ISBN 0-7356-2199-3

One of the key features of SQL Server 2005 is SQL Server Analysis Servicesâ Microsoft's customizable analysis solution for business data modeling and interpretation. Just compare SQL Server Analysis Services to its competition to understand the great value of its enhanced features. One of the keys to harnessing the full functionality of SQL Server will be leveraging Analysis Services for the powerful tool that it isâ including creating a cube, and deploying, customizing, and extending the basic calculations. This step-by-step tutorial discusses how to get started, how to build scalable analytical applications, and how to use and administer advanced features. Interactivity (enhanced in SQL Server 2005), data translation, and security are also covered in detail.

Microsoft SQL Server 2005 Reporting Services Step by Step
 Hitachi Consulting Services ª ISBN 0-7356-2250-7

SQL Server Reporting Services (SRS) is Microsoft's customizable reporting solution for business data analysis. It is one of the key value features of SQL Server 2005: functionality more advanced and much less expensive than its competition. SRS is powerful, so an understanding of how to architect a report, as well as how to install and program SRS, is key to harnessing the full functionality of SQL Server. This procedural tutorial shows how to use the Report Project Wizard, how to think about and access data, and how to build queries. It also walks through the creation of charts and visual layouts for maximum visual understanding of data analysis. Interactivity (enhanced in SQL Server 2005) and security are also covered in detail.

Programming Microsoft SQL Server 2005
 Andrew J. Brust, Stephen Forte, and William H. Zack
 ISBN 0-7356-1923-9

This thorough, hands-on reference for developers and database administrators teaches the basics of programming custom applications with SQL Server 2005. You will learn the fundamentals of creating database applicationsâ including coverage of T-SQL, Microsoft .NET Framework, and Microsoft ADO.NET. In addition to practical guidance on database architecture and design, application development, and reporting and data analysis, this essential reference guide covers performance, tuning, and availability of SQL Server 2005.

Inside Microsoft SQL Server 2005: The Storage Engine
 Kalen Delaney ª ISBN 0-7356-2105-5

Inside Microsoft SQL Server 2005: T-SQL Programming
 Itzik Ben-Gan ª ISBN 0-7356-2197-7

Inside Microsoft SQL Server 2005: Query Processing and Optimization
 Kalen Delaney ª ISBN 0-7356-2196-9

Programming Microsoft ADO.NET 2.0 Core Reference
 David Sceppa ª ISBN 0-7356-2206-X

For more information about Microsoft Press® books and other learning products, visit:
www.microsoft.com/mspress and www.microsoft.com/learning



Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at www.microsoft.com/mspress. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call 1-800-268-2222.)

Prepare for Certification with Self-Paced Training Kits:



Ace your preparation for the skills measured by the MCP exams—and on the job. With official Self-Paced Training Kits from Microsoft, you'll work at your own pace through a system of lessons, hands-on exercises, troubleshooting labs, and review questions. Then test yourself with the Readiness Review Suite on CD, which provides hundreds of challenging questions for in-depth self-assessment and practice.

- MCSE Self-Paced Training Kit (Exams 70-290, 70-291, 70-293, 70-294): Microsoft® Windows Server™ 2003 Core Requirements. 4-Volume Boxed Set. ISBN: 0-7356-1953-0. (Individual volumes are available separately.)
- MCSA/MCSE Self-Paced Training Kit (Exam 70-270): Installing, Configuring, and Administering Microsoft Windows® XP Professional, Second Edition. ISBN: 0-7356-2152-7.
- MCSE Self-Paced Training Kit (Exam 70-298): Designing Security for a Microsoft Windows Server 2003 Network. ISBN: 0-7356-1969-7.
- MCSA/MCSE Self-Paced Training Kit (Exam 70-350): Implementing Microsoft Internet Security and Acceleration Server 2004. ISBN: 0-7356-2169-1.
- MCSA/MCSE Self-Paced Training Kit (Exam 70-284): Implementing and Managing Microsoft Exchange Server 2003. ISBN: 0-7356-1899-2.

For more information about Microsoft Press® books, visit: www.microsoft.com/mspress

For more information about learning tools such as online assessments, e-learning, and certification, visit:
www.microsoft.com/mspress and www.microsoft.com/learning



Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at www.microsoft.com/mspress. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call 1-800-268-2222.)

2007 Microsoft® Office System Resources for Developers and Administrators



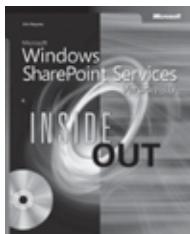
Microsoft Office SharePoint® Server 2007 Administrator's Companion

Bill English with the Microsoft

SharePoint Community Experts

ISBN 9780735622821

Get your mission-critical collaboration and information management systems up and running. This comprehensive, single-volume reference details features and capabilities of SharePoint Server 2007. It delivers easy-to-follow procedures, practical workarounds, and key troubleshooting tactics— for on-the-job results.



Microsoft Windows SharePoint Services Version 3.0 Inside Out

Erin O'Connor

ISBN 9780735623231

Conquer Microsoft Windows SharePoint Services— from the inside out! This ultimate, in-depth reference packs hundreds of time-saving solutions, troubleshooting tips, and workarounds. You're beyond the basics, so now learn how the experts tackle information sharing and team collaboration—and challenge yourself to new levels of mastery!



Microsoft SharePoint Products and Technologies Administrator's Pocket Consultant

Ben Curry

ISBN 9780735623828

Portable and precise, this pocket-sized guide delivers immediate answers for the day-to-day administration of SharePoint Products and Technologies. Featuring easy-to-scan tables, step-by-step instructions, and handy lists, this book offers the straightforward information you need to get the job done— whether you're at your desk or in the field!



Inside Microsoft Windows® SharePoint Services Version 3

Ted Pattison and Daniel Larson

ISBN 9780735623200

Get in-depth insights on Microsoft Windows SharePoint Services with this hands-on guide. You get a bottom-up view of the platform architecture, code samples, and task-oriented guidance for developing custom applications with Microsoft Visual Studio® 2005 and Collaborative Application Markup Language (CAML).

Inside Microsoft Office SharePoint Server 2007

Patrick Tisseghem

ISBN 9780735623682

Dig deepâ and master the intricacies of Office SharePoint Server 2007. A bottom-up view of the platform architecture shows you how to manage and customize key components and how to integrate with Office programsâ helping you create custom enterprise content management solutions.

Microsoft Office Communications Server 2007 Resource Kit

Microsoft Office Communications Server Team

ISBN 9780735624061

Your definitive reference to Office Communications Server 2007â direct from the experts who know the technology best. This comprehensive guide offers in-depth technical information and best practices for planning, designing, deploying, managing, and optimizing your systems. Includes a toolkit of valuable resources on CD.

Programming Applications for Microsoft Office Outlook® 2007

Randy Byrne and Ryan Gregg

ISBN 9780735622494

Microsoft Office Visio® 2007 Programming Step by Step

David A. Edson

ISBN 9780735623798

See more resources at microsoft.com/mspress and microsoft.com/learning

**Microsoft®
Press**

Microsoft Press® products are available worldwide wherever quality computer books are sold. For more information, contact your bookseller, computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at microsoft.com/mspress. To locate a source near you, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call 1-800-268-2222.)

Additional Resources for Developers from Microsoft Press

[Visual Basic 2005](#)

[Visual C# 2005](#)

[Web Development](#)

[Data Access](#)

[SQL Server 2005](#)

[Other Developer Topics](#)

Visual Basic 2005

Microsoft Visual Basic® 2005 Express Edition: Build a Program Now!

Patrice Pelland

978-0-7356-2213-5

Microsoft Visual Basic 2005 Step by Step

Michael Halvorson

978-0-7356-2131-2

Programming Microsoft Visual Basic 2005: The Language

Francesco Balena

978-0-7356-2183-1

Additional Resources for Developers from Microsoft Press

[Visual Basic 2005](#)

[Visual C# 2005](#)

[Web Development](#)

[Data Access](#)

[SQL Server 2005](#)

[Other Developer Topics](#)

Visual Basic 2005

Microsoft Visual Basic® 2005 Express Edition: Build a Program Now!

Patrice Pelland

978-0-7356-2213-5

Microsoft Visual Basic 2005 Step by Step

Michael Halvorson

394

978-0-7356-2131-2

Programming Microsoft Visual Basic 2005: The Language

Francesco Balena

978-0-7356-2183-1

Visual C# 2005

Microsoft Visual C#® 2005 Express Edition: Build a Program Now!

Patrice Pelland

978-0-7356-2229-6

Microsoft Visual C# 2005 Step by Step

John Sharp

978-0-7356-2129-9

Programming Microsoft Visual C# 2005: The Language

Donis Marshall

978-0-7356-2181-7

Programming Microsoft Visual C# 2005: The Base Class Library

Francesco Balena

978-0-7356-2308-8

CLR via C#, Second Edition

Jeffrey Richter

978-0-7356-2163-3

Microsoft .NET Framework 2.0 Poster Pack

Jeffrey Richter

978-0-7356-2317-0

Web Development

Microsoft Visual Web Developer® 2005 Express Edition: Build a Web Site Now!

Jim Buyens

978-0-7356-2212-8

Microsoft ASP.NET 2.0 Step by Step

George Shepherd

978-0-7356-2201-2

Programming Microsoft ASP.NET 2.0 Core Reference

Dino Esposito

978-0-7356-2176-3

Programming Microsoft ASP.NET 2.0 Applications Advanced Topics

Dino Esposito

978-0-7356-2177-0

Developing More-Secure Microsoft ASP.NET 2.0 Applications

Dominick Baier

978-0-7356-2331-6

Data Access

Microsoft ADO.NET 2.0 Step by Step

Rebecca M. Riordan

978-0-7356-2164-0

Programming Microsoft ADO.NET 2.0 Core Reference

David Sceppa

978-0-7356-2206-7

Programming Microsoft ADO.NET 2.0 Applications Advanced Topics

Glenn Johnson

978-0-7356-2141-1

SQL Server 2005

Microsoft SQL Server 2005 Database Essentials Step by Step

Solid Quality Learning

978-0-7356-2207-4

Microsoft SQL Server 2005 Applied Techniques Step by Step

Solid Quality Learning

978-0-7356-2316-3

Microsoft SQL Server 2005 Analysis Services Step by Step

Reed Jacobson, Stacia Misner, and Hitachi Consulting

978-0-7356-2199-2

Microsoft SQL Server 2005 Reporting Services Step by Step

Stacia Misner Hitachi Consulting

978-0-7356-2250-0

Microsoft SQL Server 2005 Integration Services Step by Step

Paul Turley Hitachi Consulting

978-0-7356-2405-4

Programming Microsoft SQL Server 2005

Andrew J. Brust Stephen Forte

978-0-7356-1923-4

Inside Microsoft SQL Server 2005: The Storage Engine

Kalen Delaney

978-0-7356-2105-3

Inside Microsoft SQL Server 2005: T-SQL Programming

Itzik Ben-Gan, Dejan Sarka, and Roger Wolter

978-0-7356-2197-8

Inside Microsoft SQL Server 2005: T-SQL Querying

Itzik Ben-Gan, Lubor Kollar, and Dejan Sarka

396

978-0-7356-2313-2

Inside Microsoft SQL Server 2005: Query Tuning and Optimization

Kalen Delaney, et al.

978-0-7356-2196-1

Other Developer Topics

Debugging Microsoft .NET 2.0 Applications

John Robbins

978-0-7356-2202-9

Hunting Security Bugs

Tom Gallagher, Bryan Jeffries, and Lawrence Landauer

978-0-7356-2187-9

Software Estimation: Demystifying the Black Art

Steve McConnell

978-0-7356-0535-0

The Security Development Lifecycle

Michael Howard Steve Lipner

978-0-7356-2214-2

Writing Secure Code, Second Edition

Michael Howard David LeBlanc

978-0-7356-1722-3

Code Complete, Second Edition

Steve McConnell

978-0-7356-1967-8

Software Requirements, Second Edition

Karl E. Wiegers

978-0-7356-1879-4

More About Software Requirements: Thorny Issues and Practical Advice

Karl E. Wiegers

978-0-7356-2267-8

microsoft.com/mspress

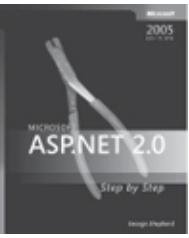
More Great Developer Resources from Microsoft Press

[Developer Step by Step](#)

[Developer Reference](#)

[Focused Topics](#)

Developer Step by Step

<ul style="list-style-type: none"> • Hands-on tutorial covering fundamental techniques and features • Practice files on CD • Prepares and informs new-to-topic programmers 	 Microsoft® Visual Basic® 2005 Step by Step Michael Halvorson 978-0-7356-2131-2	 Microsoft Visual C#® 2005 Step by Step John Sharp 978-0-7356-2129-9	 Microsoft ADO.NET 2.0 Step by Step Rebecca M. Riordan 978-0-7356-2164-0	 Microsoft ASP.NET 2.0 Step by Step George Shepherd 978-0-7356-2201-2
---	---	--	---	---

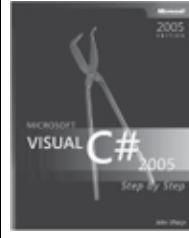
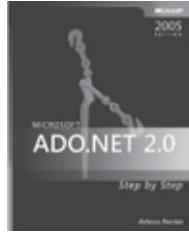
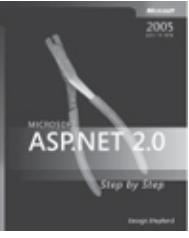
More Great Developer Resources from Microsoft Press

[Developer Step by Step](#)

[Developer Reference](#)

[Focused Topics](#)

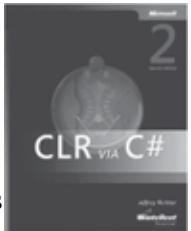
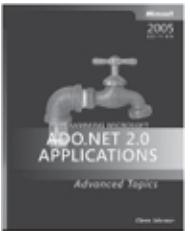
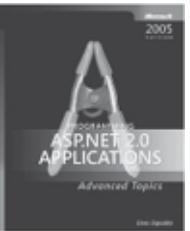
Developer Step by Step

<ul style="list-style-type: none"> • Hands-on tutorial covering fundamental techniques and features • Practice files on CD • Prepares and informs new-to-topic programmers 	 Microsoft® Visual Basic® 2005 Step by Step Michael Halvorson 978-0-7356-2131-2	 Microsoft Visual C#® 2005 Step by Step John Sharp 978-0-7356-2129-9	 Microsoft ADO.NET 2.0 Step by Step Rebecca M. Riordan 978-0-7356-2164-0	 Microsoft ASP.NET 2.0 Step by Step George Shepherd 978-0-7356-2201-2
---	---	--	---	---

Developer Reference

<ul style="list-style-type: none"> • Expert coverage of core topics • Extensive, pragmatic coding examples • Builds professional proficiency with a Microsoft technology 	 Programming Microsoft Visual Basic 2005: The Language Francesco Balena 978-0-7356-2183-1	 Programming Microsoft Visual C# 2005: The Language Donis Marshall 978-0-7356-2181-7	 Programming Microsoft ADO.NET 2.0 Core Reference David Sceppa 978-0-7356-2206-7	 Programming Microsoft ASP.NET 2.0 Core Reference Dino Esposito 978-0-7356-2176-3
---	---	--	---	---

Focused Topics

<ul style="list-style-type: none"> • Deep coverage of advanced techniques and capabilities • Extensive, adaptable coding examples • Promotes full mastery of a Microsoft technology 	 CLR via C#, Second Edition Jeffrey Richter 978-0-7356-2163-3	 Debugging Microsoft .NET 2.0 Applications John Robbins 978-0-7356-2202-9	 Programming Microsoft ADO.NET 2.0 Applications Advanced Topics Glenn Johnson 978-0-7356-2141-1	 Programming Microsoft ASP.NET 2.0 Applications Advanced Topics Dino Esposito 978-0-7356-2177-0
--	--	--	---	--

See even more titles on our Web site!

Explore our full line of learning resources at: microsoft.com/mspress and microsoft.com/learning

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

- Adhoc caching
- Adhoc query caching 2nd
- Affinity mask 2nd 3rd 4th 5th
- Aggregation placement
- Aggregations
- Allocation bottleneck
- ALTER TRACE permission
- ANSI-style join hints
- Application architecture
 - application program interface (API)
 - application server
 - business logic
 - caching
 - client tier
 - connection pooling
 - data tier
 - middle/application tier
 - multiple servers, using
 - pooling connections
 - three tiers
 - two-tier model
 - web server
- Application design
 - application
 - data manipulation language (DML)
 - database schema and its physical design
 - logical unit numbers (LUN)
 - mapping heavily accessed tables onto the same physical disk
 - mapping to physical disks
 - mixing log and data onto the same physical disk
 - sharing physical disk among tempdb and user databases
 - storage area network (SAN) environment
 - useful indexes, availability of
- Application program interface (API)
- Application server
- Atomicity
- Auditing
 - blackbox trace
 - C2 and common criteria auditing
 - default trace
- Auto and explicit growth of tempdb
- AUTO_CREATE_STATISTICS
- AUTO_UPDATE_STATISTICS
- Autoparameterization 2nd 3rd
- Available bytes
- Average Wait Time (ms)
- Avg. Disk Queue Length

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Baseline for your workload, creating a
 Bitmap filtering
 Blackbox trace
 Blocked process report
 Blocking
 Average Wait Time (ms)
 blocked process report
 DBCC SQLPERF
 deadlocks
 detecting blocking problems
 detection of blocking
 finding the cause of blocking
 isolating and troubleshooting
 joining sys.dm_os_waiting_tasks and sys.dm_tran_locks
 killing a session
 LAZYWRITER_SLEEP
 LCK_M_S
 lock escalation
 Lock Request/Sec
 Lock Wait Time (ms)
 Lock Waits/Sec
 Number of Deadlocks/Sec
 overview
 PAGEIOLATCH_EX
 PAGEIOLATCH_SH
 Processes Blocked
 READ COMMITTED
 READ UNCOMMITTED 2nd
 READ_COMMITTED_SNAPSHOT
 resolving blocking problems
 resolving reader/writer blocking
 resolving writer/writer blocking
 SELECT
 signal wait
 SNAPSHOT ISOLATION
 SQLDiag utility
 sys.dm_os_waiting_tasks DMV
 sys.dm_tran_locks DMV
 system monitor counters
 Table Lock Escalations/Sec
 Traceflag 1221
 Traceflag 1224
 Blocking detection tool
 Blocking resolution tool

Bookmark lookup 2nd
 Branch
 Broadcast partitioning
 Buffer Cache hit ratio
 Buffer cache hit ratio
 Business logic

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

C2 and common criteria auditing
 Cache bytes
 Cache size management
 Cache stores
 Cache-hit ratio
 CachedPlanSize
 Cacheobjtype 2nd
 Caching
 Caching mechanisms 2nd
 Caching prepared queries
 Cardinality estimation errors
 Cascading deadlocks
 CASE expressions
 Checkpoint pages/sec
 Clearing plan cache
 Client tier
 Close method 2nd
 CLR UDF
 Clustered index 2nd 3rd
 CMEMTHREAD
 Colmodctr values
 Column order
 Column statistics
 Columns
 Commit limit
 Committed
 Common Table Expression (CTE)
 Compilation and optimization 2nd
 Compilation and recompilation, excessive
 CompileCPU
 Compiled objects
 Compiled plans
 CompileMemory
 CompileTime
 Complexity, avoid unnecessary
 Composite indexes
 CONCAT UNION
 Concurrency issues
 Connection pooling
 Connection settings

Consistency
 Constraints
 Conversion deadlock
 Correctness-based recompiles
 Correlated scalar subqueries
 Cost threshold for parallelism 2nd
 Costing
 Costing of cache entries
 Covered columns 2nd
 Covering nonclustered index
 CPU bottlenecks
 compilation and recompilation, excessive detection
 inefficient query plan
 isolating and troubleshooting optimizations
 recompile options
 runnable
 running
 schema change
 set option
 suspended
 total pages
 update statistics
 CPU-Related configuration options
CREATE INDEX
CREATE LOGIN
CREATE STATISTICS

Index

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

Data access for tables with different index structures
 Data collection
 Data manipulation language (DML)
 Data tier
 Database Engine Tuning Advisor (DTA) 2nd
 Database engine tuning advisor (DTA)
 DBCC commands
 DBCC INPUTBUFFER
 DBCC MEMORYSTATUS
 DBCC SQLPERF
 DDL bottleneck 2nd
 DDL conflicts
 Deadlocking
 cascading deadlocks
 conversion deadlock
 detecting deadlocks
 determining the cause of deadlocks
 lock-based deadlocking

potential deadlocks
 reader/writer cycle deadlocks
 reader/writer deadlocks
 resolving
 SQL trace deadlock event classes
 trace flags 1204
 trace flags 1222 2nd
 types of
 writer/writer cycle deadlock
 writer/writer deadlocks
 Deadlocks 2nd
 Default trace
 Degree of parallelism (DOP) 2nd
DELETE
 Disk Reads/Sec or Disk Writes/Sec
 Distinct aggregates
 Durability
 Dynamic cursor support
 Dynamic management views (DMV) and functions (DMF)
 Dynamic SQL
 Dynamically generated SQL statements

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Estimated vs. actual query plans
 Event classes, common
 Event producers
 Events
 Events selection tab
 EventSequence 2nd 3rd
 Exceptions, identifying
 Exclusive (X) lock
 Execution plans
 EXPAND VIEWS hint

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

FAST 1 2nd
 FAST 100 2nd
 FAST N hint
FASTFIRSTROW
 FASTFIRSTROW hint
 File provider

[Filter on duration](#)
[Filters](#)
[Filtration](#)
[First implementation](#)
[Fixing RI triggers](#)
[Fn_trace_geteventinfo](#)
[Fn_trace_getfilterinfo](#)
[Fn_trace_gettable](#)
[FORCE ORDER 2nd](#)
[Forced parameterization](#)
[Foreign key lookups and snapshot-based isolation levels](#)
[Free space in tempdb \(KB\)](#)
[Full outer joins](#)
[Full SNAPSHOT isolation, use cases for](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[GAM pages](#)
[General tab](#)
[GetRow method 2nd](#)
[Global memory pressure](#)
[GRANT](#)
[Graphical plans](#)
[GROUP BY 2nd 3rd](#)
[GROUP hints](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Handles](#)
[Hardware resources](#)
[Hash aggregation 2nd](#)
[HASH GROUP](#)
[HASH JOIN](#)
[Hash join 2nd](#)
[Hash partitioning](#)
[HASH UNION](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

I/O bottlenecks
 Avg. Disk Queue Length
 Avg. Disk Sec/Read or Avg. Disk Sec/Write
 detection of
 Disk Reads/Sec or Disk Writes/Sec
 isolating and troubleshooting
 PAGEIOLATCH_EX
 PAGEIOLATCH_SH
 RUNNABLE state
 SUSPENDED state
 I/O provider
 Identifying an index's keys
 In order option
 INDEX
 Index columns used in joins
 Index cost
 INDEX hint
 Index intersections
 Index node page
 Index selection
 Index statistics
 Index tuning wizard (ITW)
 Index unions
 Indexes, creating useful
 Inefficient query plan
 Inline table-valued functions (TVFs)
 Inner-side parallel execution
 INSERT
 Inserts, updates, and deletes
 Internal trace components
 Isolation
 Isolation level
 Iterators

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Join order
 Join predicated and logical join types 2nd
 Joins

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

[KEEP PLAN](#)
[KEEPFIXED PLAN](#)
[Killing a session](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[LAZYWRITER_SLEEP](#)
[Lazywrites/Sec](#)
[LCK_M_S](#)
[Leaf level](#)
[Left deep vs. right deep vs. bushy hash join trees](#)
[Lightweight pooling](#)
[Load balancing](#)
[Local memory pressure](#)
[Lock escalation 2nd](#)
[Lock memory \(KB\) counter](#)
[Lock Request/Sec](#)
[Lock timeout](#)
[Lock Wait Time \(ms\)](#)
[Lock Waits/Sec](#)
[Lock-based deadlocking](#)
[Locked pages](#)
[Locking 2nd](#)

- lock escalation
- lock memory (KB) counter
- lock timeout
- resolving lock escalation
- troubleshooting lock memory

[Locking granularity](#)
[Logical I/O](#)
[Logical reads](#)
[Logical unit numbers \(LUN\)](#)
[LOOP JOIN](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Manual statistics refresh](#)
[Mapping heavily accessed tables onto the same physical disk](#)
[Mapping to physical disks](#)
[Max and min server memory](#)
[Max degree of parallelism 2nd 3rd](#)
[Max file size](#)
[Max worker threads 2nd 3rd](#)

MAXDOP N 2nd
 Memory and spilling 2nd
 Memory bottlenecks
 Buffer Cache hit ratio
 checkpoint pages/sec
 committed
 detection of physical memory pressure
 lazywrites/Sec
 memory: available bytes
 multipage allocation
 overview
 page life expectancy
 physical memory pressure
 single-page allocations
 target
 virtual memory pressure
 Memory consumption
 Memory pressure
 Memory-related configuration options
 Memory: available bytes
 Memory_object_address
 MemoryGrant
 MERGE JOIN 2nd 3rd 4th
 MERGE UNION 2nd
 Middle/application tier
 Mixing log and data onto the same physical disk
 Modification counters
 Monitoring row versioning
 Monitoring snapshot isolation options
 Monitoring the workload
 Multicolumn statistics
 Multipage allocation
 Multiple Active Row Sets (MARS)
 Multiple distinct 2nd
 Multiple indexes
 Multiple nonclustered indexes
 Multiple plans in cache
 Multiple recompilations
 Multiple servers, using
 Multiple threads, replay using 2nd

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Nested loops join 2nd
 NO EXPAND 2nd
 Nonblocking vs. blocking iterators
 Nonclustered index 2nd
 Nonclustered index on a table with a clustered index
 Nonclustered index seek on a heap

[Noncorrelated scalar subqueries](#)

[Number of Deadlocks/Sec](#)

[Number of replay threads](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Object plan guides](#)

[Objects in plan cache](#)

[One-to-many vs. many-to-many merge join](#)

[Online index build](#)

[Open method 2nd](#)

[Optimality-based recompiles](#)

[Optimization](#)

[Optimization hints 2nd](#)

[Optimization level](#)

[Optimizations](#)

[OPTIMIZE FOR](#)

[OPTIMIZE FOR hint](#)

[ORDER BY](#)

[ORDER GROUP](#)

[OUTER PREFERENCES](#)

[Overview](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Page life expectancy 2nd](#)

[Page split counter](#)

[PAGEIOLATCH_EX 2nd](#)

[PAGEIOLATCH_SH 2nd](#)

[Paging file: %usage](#)

[Parallel deadlocks](#)

[Parallel hash join](#)

[Parallel merge join](#)

[Parallel nested loops join](#)

[Parallel nested loops join performance](#)

[Parallel scan](#)

[Parallelism](#)

[affinity mask](#)

[affinity mask64](#)

[application](#)

[cost threshold for parallelism](#)

[degree of parallelism \(DOP\)](#)

[inner-side parallel execution](#)

load balancing
 max degree of parallelism
MAXDOP N
 parallel deadlocks
 parallel hash join
 parallel merge join
 parallel nested loops join
 parallel nested loops join performance
 parallel scan
 parallelism exchange operators
 parallelism operator
 partitioning for executing parallel queries
 pipeline parallelism
 round robin exchange
PARAMETERIZATION SIMPLE/FORCED
 Partitioned tables
 Partitioning for executing parallel queries
 Performance monitor
 Performance tuning
 PFS pages
 Physical memory pressure
 Physical reads
 Pipeline parallelism
 Plan cache
 adhoc caching
 adhoc query caching 2nd
 autoparameterization 2nd 3rd
 cache size management
 cache stores
 cacheobjtype 2nd
 caching mechanisms 2nd
 caching prepared queries
 clearing plan cache
CMEMTHREAD
 compiled objects
 compiled plans
 correctness-based recompiles
 costing of cache entries
 execution plans
 forced parameterization
 functions
 global memory pressure
 handles
 internals
KEEP PLAN
KEEPFIXED PLAN
 local memory pressure
 memory_object_address
 multiple plans in cache
 multiple recompilations
 objects in plan cache
 optimality-based recompiles
 optimization hints
OPTIMIZE FOR

PARAMETERIZATION simple/forced
 plan cache metadata 2nd
 plan guides
 prepare and execute method 2nd
 prepared queries
 recompilation
RECOMPILE
 removing plans from cache
RESOURCE_SEMAPHORE_QUERY_COMPILE
 role of
 size_in_bytes
 skipping recompilation step
SOS_RESERVEDMEMBLOCK LIST
 sp_executesql procedure 2nd
SQL trace
 stored procedures 2nd
 sys.dm_exec_cached_plan_dependent_objects
 sys.dm_exec_cached_plans
 sys.dm_exec_query_stats
 sys.dm_exec_requests
 sys.dm_exec_sql_text
 syscacheobjects
 system monitor (Perfmon)
 troubleshooting
USE PLAN
 wait statistics
 Plan cache metadata 2nd
 Plan guides
 considerations
 managing of
 object plan guides
 purpose of
 SQL plan guides
 template plan guides
 types of
 Pool nonpaged bytes
 Pooling connections
 Potential SNAPSHOT isolation level conflicts
 Prepare and execute method 2nd
 Prepared queries
 Preventing lost updates
PRIMARY KEY
 Private bytes 2nd
 Processes Blocked
 Profiler
 basics
 events selection tab
 EventSequence 2nd 3rd
 filter on duration
 filtration
 fn_trace_geteventinfo
 fn_trace_getfilterinfo
 fn_trace_gettable
 general tab

in order option
multiple threads, replay using 2nd
number of replay threads
querying server-side trace metadata
questionable usage of
replay events in the order they were traced
replay options dialog box 2nd
retrieving data from server-side traces
rowset provider, investigating
save to table option
saving files
scripting server-side traces
server-side tracing and collection
sp_trace_create 2nd
sp_trace_setevent
sp_trace_setfilter
sp_trace_setstatus 2nd
SQL:BatchCompleted
StartTime/EndTime
sys.trace_columns
sys.trace_events
sys.traces catalog view 2nd
template
trace file option
trace table option
trace xml file for replay option
trace XML file option
traces, saving and replaying
traces, stopping and closing
TSQL_Replay 2nd

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

[Query analysis](#)

Query hints

ANSI-style join hints
bookmark lookup
EXPAND VIEWS hint
FAST N hint
FASTFIRSTROW hint
FORCE ORDER
GROUP hints
INDEX hint
MAXDOP N hint
multiple nonclustered indexes
NO EXPAND hints
OPTIMIZE FOR hint
query-level join hints
table hints

UNION hints

USE PLAN hint

Query improvements

indexes, creating useful

optimization hints

rewriting your query

schema improvements

statistics management

Query optimizer

Query performance, monitoring

Query plan display options

Query processing and execution

aggregations

bookmark lookup

CachedPlanSize

Close method 2nd

CompileCPU

CompileMemory

CompileTime

composite indexes

costing

covered columns 2nd

degree of parallelism

dynamic cursor support

estimated vs. actual query plans

full outer joins

GetRow method 2nd

graphical plans

hash aggregation 2nd

hash join 2nd

identifying and index's keys

index intersections

index unions

inserts, updates, and deletes

iterators

join predicated and logical join types 2nd

joins

left deep vs. right deep vs. bushy hash join trees

memory and spilling 2nd

memory consumption

MemoryGrant

merge join

nested loops join

nonblocking vs. blocking iterators

one-to-many vs. many-to-many merge join

Open method 2nd

parallelism

partitioned tables

properties of iterators

query plan display options

QueryPlan

reading query plans

RelOp

scalar aggregation

scans and seeks
 seekable predicates
 SET SHOWPLAN_ALL ON
 SET SHOWPLAN_TEXT ON 2nd
 single-column indexes
 sort merge join vs. index merge join
 StatementSetOptions
 StmtSimple
 stream aggregation
 subqueries
 text plans
 Query subtree
 Query-level join hints
 Querying server-side trace metadata
 QueryPlan

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Read ahead reads
 READ COMMITTED 2nd
 READ COMMITTED SNAPSHOT 2nd
 READ COMMITTED SNAPSHOT ISOLATION
 READ UNCOMMITTED 2nd 3rd
 READ_COMMITTED_SNAPSHOT
 Reader/writer cycle deadlocks
 Reader/writer deadlocks
 Reading query plans
 Recompilation
 RECOMPILE
 Reducing trace overhead
 RelOp
 Removing correlations
 Removing plans from cache
 Repeatable read
 Replay events in the order they were traced
 Replay options dialog box 2nd
 Resolving blocking problems
 Resolving lock escalation
 Resolving snapshot isolation problems
 RESOURCE_SEMAPHORE_QUERY_COMPILE
 Response time
 Retrieving data from server-side traces
 Retrying transactions to resolve update conflicts
 Rewriting your query
 Rogue session
 Rollover files
 Round robin exchange
 Row-versioning-based isolation levels
 appropriate uses for snapshot isolation

DDL conflicts
 dynamic management views
 fixing RI triggers
 foreign key lookups and snapshot-based isolation levels
 full SNAPSHOT isolation, use cases for
 monitoring row versioning
 monitoring snapshot isolation options
 page split counter
 potential SNAPSHOT isolation level conflicts
 preventing lost updates
READ COMMITTED SNAPSHOT 2nd
 resolving snapshot isolation problems
 retrying transactions to resolve update conflicts
 row versioning issues, resolving
 shared locking issues
 snapshot isolation and triggers
 SNAPSHOT isolation level issues, resolving
 tempdb issues caused by snapshot isolation, resolving
 tempdb, monitoring
 trigger-based RI and snapshot isolation
 triggers and the OUTPUT clause
 troubleshooting
 update conflicts and full SNAPSHOT isolation
 update conflicts counter
 update conflicts: potential lost updates
 version cleanup rate
 version generation rate
 version store counters
 version store size
 version store unit count
 version store unit creation
 version store unit truncation
Rowset provider 2nd
Runnable
RUNNABLE state
Running

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Save to table option
 Saving files
 Scalability
 Scalar aggregation
 Scalar distinct
 Scan count
 Scans and seeks
 Schema change
 Schema improvements
 Scripting server-side traces

Search argument (SARG)
Second implementation
Secondary spools
Security and permissions
Seekable predicates
Segment spool
SELECT
Select distinct
SELECT INTO
Sensitive event data, protecting
Serializable
SERIALIZABLE isolation
SERT STATISTICS PROFILE ON
Server-side tracing and collection
Set option
SET SHOWPLAN_ALL ON
SET SHOWPLAN_TEXT ON 2nd
Set-oriented programming
SGAM pages
Shared (s) lock
Shared locking issues
Sharing physical disk among tempdb and user databases
SHOWPLAN
Signal wait
Single-column indexes
Single-page allocations
Size_in_bytes
Skipping recompilation step
SNAPSHOT ISOLATION 2nd
Snapshot isolation and triggers
SNAPSHOT isolation level issues, resolving
Sort merge join vs. index merge join
SOS_RESERVEDMEMBLOCKLIST
Sp_executesql procedure 2nd
Sp_trace_create 2nd
Sp_trace_generateevent
Sp_trace_setevent
Sp_trace_setfilter
Sp_trace_setstatus 2nd
SQL plan guides
SQL server
 application
 definition
SQL server configuration
 affinity mask
 AWE enabled
 CPU-Related configuration options
 lightweight pooling
 max and min server memory
 max degree of parallelism
 max worker threads
 memory-related configuration options
SQL server profiler
SQL trace

SQL Trace architecture

- ALTER TRACE permission
- columns
- event producers
- events
- file provider
- filters
- I/O provider
- internal trace components
- overview
- rowset provider
- security and permissions
- sensitive event data, protecting
- trace controller
- trace I/O providers
- SQL trace deadlock event classes
- SQL:BatchCompleted
- SQLDiag utility
- Stale statistics
- StartTime/EndTime
- StatementSetOptions
- Statistics
- STATISTICS IO 2nd
- Statistics management
- Statistics on computed columns
- STATISTICS PROFILE 2nd
- STATISTICS TIME 2nd
- STATISTICS XML 2nd
- Step
- StmtSimple
- Storage area network (SAN) environment
- Stored procedures 2nd 3rd
- Stream aggregation
- Subqueries
 - CASE expressions
 - correlated scalar subqueries
 - noncorrelated scalar subqueries
 - removing correlations
- Suspended
- SUSPENDED state
- Sys.dm_exec_cached_plan_dependent_objects
- Sys.dm_exec_cached_plans
- Sys.dm_exec_query_stats
- Sys.dm_exec_requests
- Sys.dm_exec_sql_text
- Sys.dm_os_waiting_tasks
- Sys.dm_os_waiting_tasks and sys.dm_tran_locks
- Sys.dm_os_waiting_tasks DMV
- Sys.dm_tran_locks DMV
- Sys.trace_columns
- Sys.trace_events
- Sys.traces catalog view 2nd
- Syscacheobjects
- System monitor (Perfmon) 2nd

[System monitor counters](#)

Index

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Table hints

Table Lock Escalations/Sec

Target

Temp Tables Creation Rate

Temp Tables for Destruction

Tempdb

 growth

 issues caused by snapshot isolation, resolving

 monitoring

 performance issues

 usage

Tempdb bottlenecks

 allocation bottleneck

 auto and explicit growth of tempdb

 CREATE INDEX

 CREATE STATISTICS

 DDL bottleneck 2nd

 GAM pages

 isolation and troubleshooting

 Multiple Active Row Sets (MARS)

 online index build

 PFS pages

 READ COMMITTED SNAPSHOT ISOLATION

 SGAM pages

 SNAPSHOT ISOLATION

 Temp Tables Creation Rate

 Temp Tables for Destruction

 tempdb growth

 tempdb performance issues

 version store

 Workfiles Created/Sec

 Worktables Created/Sec

 Worktables from Cache Ratio

Template plan guides

Temporary tables with unqualified schema

Three tiers

Throughput

Total pages

Trace controller

Trace file option

Trace flags 1204

Trace flags 1222 2nd

Trace I/O providers

Trace table option

Trace xml file for replay option

Trace XML file option

Traceflag 1221

Traceflag 1224

Traces

CLR UDF

considerations and design

data collection

deadlocks, debugging

event classes, common

exceptions, identifying

max file size

performance tuning

reducing trace overhead

rollover files

saving and replaying

sp_trace_generateevent

stored procedure debugging

troubleshooting and analysis

TRACEWRITE

Transact-SQL code

Transactions and isolation levels

application

atomicity

consistency

durability

exclusive (X) lock

first implementation

isolation

read committed

read uncommitted

repeatable read

second implementation

serializable

shared (S) lock

Trigger-based RI and snapshot isolation

Triggers and the OUTPUT clause

Troubleshooting

baseline for your workload, creating a

buffer cache hit ratio

database engine tuning advisor (DTA)

DBCC commands

dynamic management views (DMV) and functions (DMF)

free space in tempdb (KB)

index tuning wizard (ITW)

monitoring the workload

page life expectancy

performance monitor

SQL server profiler

system monitor

Troubleshooting concurrency

TSQL_Replay 2nd

Two-tier model

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

UNION

UNION ALL

UNION hints

UNIQUE

Unqualified object names

Unusable statistics

UPDATE

Update conflicts

Update conflicts and full SNAPSHOT isolation

Update conflicts counter

Update conflicts: potential lost updates

Update statistics

USE PLAN 2nd 3rd

Useful indexes, availability of

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Version cleanup rate

Version generation rate

Version store

Version store counters

Version store size

Version store unit count

Version store unit creation

Version store unit truncation

Virtual memory pressure 2nd

Index

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]

Wait statistics

Web server

WITH PASSWORD

Workfiles Created/Sec

Working set

Worktables Created/Sec

Worktables from Cache Ratio

Writer/writer cycle deadlock

Writer/writer deadlocks

