Microsoft

Introducing Microsoft® SQL Server® 2008

Peter DeBetta
Greg Low
Mark Whitehorn

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

Copyright © 2008 by Microsoft Corporation

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: 2007939307

Printed and bound in the United States of America.

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.

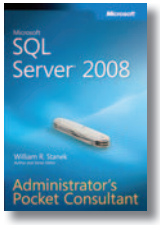
Microsoft, Microsoft Press, Active Directory, Excel, MSDN, MultiPoint, SharePoint, SQL Server, Virtual Earth, Visual Studio, Win32, Windows, Windows PowerShell, Windows Server, and Windows Vista are either registered trademarks or trademarks of the Microsoft group of companies. Other product and company 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.

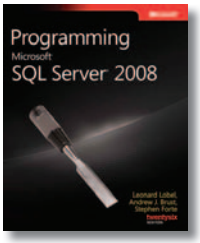
This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused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

Acquisitions Editor: Ken Jones
Developmental Editor: Sandra Haynes
Project Editor: Kathleen Atkins
Editorial Production: nSight, Inc.

More Resources for SQL Server 2008

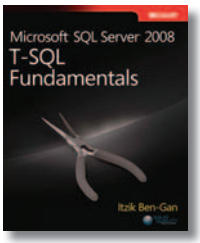


Microsoft® SQL Server® 2008 Administrator's Pocket Consultant
William R. Stanek
ISBN 9780735625891



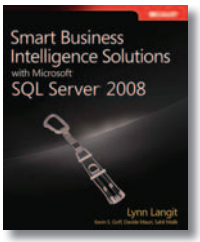
Programming Microsoft SQL Server 2008
Leonard Lobel, Andrew J. Brust, Stephen Forte
ISBN 9780735625990

Microsoft SQL Server 2008 Step by Step
Mike Hotek
ISBN 9780735626041



Microsoft SQL Server 2008 T-SQL Fundamentals
Itzik Ben-Gan
ISBN 9780735626010

MCTS Self-Paced Training Kit (Exam 70-432) Microsoft SQL Server 2008 Implementation and Maintenance
Mike Hotek
ISBN 9780735626058



Smart Business Intelligence Solutions with Microsoft SQL Server 2008
Lynn Langit, Kevin S. Goff, Davide Mauri, Sahil Malik
ISBN 9780735625808

COMING SOON

Microsoft SQL Server 2008 Internals
Kalen Delaney et al.
ISBN 9780735626249

Inside Microsoft SQL Server 2008: T-SQL Querying
Itzik Ben-Gan, Lubor Kollar, Dejan Sarka
ISBN 9780735626034

Microsoft SQL Server 2008 Best Practices
Saleem Hakani and Ward Pond with the Microsoft SQL Server Team
ISBN 9780735626225

Microsoft SQL Server 2008 MDX Step by Step
Bryan C. Smith, C. Ryan Clay, Hitachi Consulting
ISBN 9780735626188

Microsoft SQL Server 2008 Reporting Services Step by Step
Stacia Misner
ISBN 9780735626478

Microsoft SQL Server 2008 Analysis Services Step by Step
Scott Cameron, Hitachi Consulting
ISBN 9780735626201

See our complete line of books at: **microsoft.com/mspress**

Microsoft® Press

To my wife, Claudia; my son, Christopher; and my daughter, Kate—my inspiration for being the best that I can be.

—*Peter DeBetta*

Contents at a Glance

Table of Contents

Foreword

A few years ago, I began to discover our home laptop was turned off nearly every time I went to use it. I asked my wife, Claudia, to leave it on, especially in the early evening when we found ourselves using it the most. However, the trend of finding it in the off state continued. As I headed down the stairs to the living room, I discovered that she was not the culprit. I watched in awe as my then two-and-a-half-year-old son pressed the power button (which glowed an inviting blue color). He then moved the mouse until the pointer was over the even more enticing red Turn Off button on the screen, and then he clicked the mouse.

I was so proud!

Christopher hadn't been taught how to do this feat; he simply watched us and then attempted it himself, and with great success. I realized that there are some things about using a computer that are essentially innate. My daughter, Kate, who is still not quite two years old, is already trying to follow in his footsteps.

Yes, our kids have had all the usual milestones (walking, talking, and so on), but certain ones, such as shutting down Windows XP, were not on the list of things to watch for. I can't wait to see what they do next.

On to business…

Of course, learning to use SQL Server requires a little more foundation than the instinctive basics of moving and clicking a mouse; this is where learning materials such as this book come into play. I had the good fortune of being able to not only dig deep into this product, but to have access to some people who helped design and implement it. For me, learning in this manner allowed me to get some great insight. I hope this work gives you enough information and insight so that you can dig deeper into this latest release and to use it to the fullest extent. And may I suggest that you let your inner child take the reins and guide your exploration into the world of SQL Server 2008.

Acknowledgments

There are so many people who deserve kudos.

First of all, my most sincere gratitude to my wife, Claudia, and my children, Christopher and Kate, who continually give me reason to keep moving forward and to better myself. I love you all so very much.

I'd like to offer my gratitude to Drs. Greg Low and Mark Whitehorn, both of whom are experts when it comes to SQL Server, and so much more, and both of whom are contributing authors to this work—and nice fellows to boot.

Much deserved thanks go to the people at Microsoft who kept things organized and kept me in line while writing this book. This work could not happen without such a great editorial team: Ken Jones, Kathleen Atkins, Sandra Haynes, Carol Vu, Pavel Kolesnikov, Carol Whitney, Devon Musgrave, Elizabeth Hansford, Joanne Hodgins, Linda Engelman, Rosemary Caperton, Kimberly Kim, Lori Merrick, Julie Strauss, and Jennifer Brown.

Other people at Microsoft played a crucial role in the technical quality of this book. And so I offer my thanks (in no particular order) to Andrew Richardson, Bill Ramos, Torsten Grabs, Boris Baryshnikov, Buck Woody, Carolyn Chau, Chris Lee, Christian Kleinerman, Colin Lyth, Ram Ramanathan, Roni Karassik, Sean Boon, Sethu Kalavakur, Srini Acharya, T.K. Anand, Thierry D'Hers, Maria Balsamo, Xiaoyu Li, Max Verun, Matt Masson, Lin Chan, Kaloian Manassiev, Jennifer Beckmann, Il-Sung Lee, and Carl Rabeler.

Several people at Microsoft took extra time to work with me in past and recent times so that I may better understand the new technologies. To these people, I offer my gratitude (again, in no particular order): Hongfei Guo, Michael Rys, Isaac Kunen, Gert Drapers, Donald Farmer, Kevin Farlee, Dan Jones, Michael Wang, and the late and much missed Ken Henderson.

To all of my fellow bloggers at *SQLblog.com*—you have helped to create a great online resource for anyone wanting to know more about SQL Server: Aaron Bertrand, Adam Machanic (my SQLblog partner in crime), Alberto Ferrari, Alexander Kuznetsov, Allen White, Andrew Kelly, Andy Leonard, Ben Miller, Denis Gobo, Erin Welker, Greg Low, Hilary Cotter, Hugo Kornelis, James Luetkehoelter, Joe Chang, John Paul Cook, Kalen Delaney, Kent Tegels, Kevin Kline, Kirk Haselden, Lara Rubbelke, Linchi Shea, Louis Davidson, Marco Russo, Michael Rys, Michael Zilberstein, Michelle Gutzait, Mosha Pasumansky, Paul Nielsen, Richard Hundhausen, Rick Heiges, Roman Rehak, Rushabh Mehta, Sarah Henwood, and Tibor Karaszi.

I also want to thank my friends and colleagues at Wintellect, Solid Quality Mentors, and Ted Pattison Group. Many of these folks played a part, directly or indirectly, in helping me with the content of this book and with allowing me to finish this work while minimizing the time I had to spend away from my family.

And this wouldn't be complete without thanking all of my fellow colleagues and SQL Server MVPs (both past and present) who so diligently worked with the beta of SQL Server 2008. I watched conversations about the product on the newsgroups and forums and had many face-to-face chats about the new technologies. Although the list of contributors is too long to show here, I do want to mention a couple of folks who played a more active role in helping throughout the writing process: Adam Machanic, Paul Nielsen, Roman Rehak, Randy Dyess, Erin Welker, Srikkant Sridharan, Sean McCown, and Trevor Barkhouse.

Introduction

This book is about SQL Server 2008.

(Now if only it were that simple.)

Take 2…

Welcome to Microsoft SQL Server 2008 (AKA "Yukon"). Many people have been speculating that the changes from version 2000 to 2005 were more dramatic than those changes that have occurred from 2005 to 2008. Yes, SQL Server 2005 was revolutionary in many respects, but SQL Server 2008 is not without some amazing new features and capabilities.

This book is divided into seven main topics: Security and Administration, Performance, Type System Enhancements, Programmability, Storage, Enhancements for High Availability, and Business Intelligence Enhancements. Each chapter will hopefully offer you insight into the new or improved features in each of these main areas. And, although the book covers a lot of ground, it is not an exhaustive tome, and, alas, not everything new or improved is contained in this book. I leave those additional details to Books Online and fellow authors who will inevitably write more comprehensive titles.

T-SQL: Still Here to Stay

Since the integration of common language runtime (CLR)-based code into SQL Server 2005 was known on the streets, people have been speculating about its role in database development. On many occasions, I heard people speaking of T-SQL as if it was being deprecated. Even now, as the CLR integration has been enhanced, and even with the introduction of system CLR types, T-SQL is still not going anywhere—and it is still most often the best choice for retrieving and manipulating data.

Goals

The objective of this book is not to give an in-depth view of the new features of SQL Server 2008; it is a beta edition, after all, and is still subject to changes. Rather, the objective of this book is to [hopefully] help people begin to grasp what can be done with SQL Server 2008. The book is part conceptual, exploring the new features and abilities of this next generation enterprise database product, and it is part tangible, demonstrating features via C# code and a new and improved T-SQL. I hope to give you enough knowledge to get your feet wet and to explore further.

I have always been a "learn by example" kind of person, so this book is filled with a lot of samples and examples to help demonstrate the concepts. Many more examples come with SQL Server 2008. I suggest you explore, poke, and prod these examples as well.

Working with Samples

Much of the sample code in this book is designed around the various Adventure Works Cycles sample database. You can download these sample databases from *www.codeplex.com*. SQL Server 2008 Books Online has more information about these sample databases, including comparisons to both pubs and Northwind and a complete data dictionary for these sample databases.

Who Should Read This Book

Everyone should read this book, as I'm *still* trying to be the first technical author on the *New York Times* bestseller list! Since I don't really expect to make that goal, I should mention that there is an audience (albeit smaller than the millions required for the bestseller list) who could benefit from this book. This group primarily includes those people who will be involved in some capacity with a migration to SQL Server 2008 and people who currently work with SQL Server 2000 and 2005 who want to see the exciting new changes in SQL Server 2008.

So should you read this book? If you are interested in learning what new features are available in SQL Server 2008 and you want to know how to begin using these new and improved tools, I suggest this book as a starting point for that learning.

Disclaimer

As with any beta product, you should know that the things discussed in this book can change before final release. Features can be removed, added, or modified as necessary to release a solid software product.

System Requirements

This book makes use of not one but two products—SQL Server 2008 (CTP 6) and Visual Studio.NET 2008. For some of the work, you will need to have both products installed in order to run code, try examples, and so on. For a majority of the content of this book, however, an installation of SQL Server 2008 will suffice. These products are available through a variety of avenues, including MSDN Subscriptions and the Beta Programs.

You can run SQL Server 2008 on Windows Vista, Windows XP (SP1 or later), and Windows 2003. It also requires version 3.5 of the .NET Framework, so even if you do not install Visual Studio 2008, you will still be required to install the framework. Fortunately, the installation program does this for you.

Support

Every effort has been made to ensure the accuracy of this book. Microsoft Press provides support for books and companion content at the following Web site: *http://www.microsoft.com/learning/support/books*.

If you have comments, questions, or ideas regarding the materials in this book, or questions that are not answered by visiting the site just mentioned, please send them to *msinput@microsoft.com*. You can also write to us at:

Microsoft Press
Attn: Programming Microsoft Office Business Applications Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through these addresses.

Chapter 1

Security and Administration

Introduction

With all the complexity of today's IT shops and with stronger security measures being required by both the high-tech industry and by government regulations, we are going to need some new tools to help us make our way through the new world of IT. SQL Server 2005 made leaps and bounds in advances for security and management, providing the ability to more easily manage and secure servers with features such as certificate and key-based encryption and more full-featured tools. SQL Server 2008 takes these measures even further, providing the ability to more easily manage and secure database servers, with new features such as policy-based management, external key management, server and database auditing, and transparent data encryption.

Policy-Based Management

Have you ever had to ensure that only Windows logons or groups were added to Microsoft SQL Server, or that xp_cmdshell was disabled, or that no stored procedure names started with "sp_"? Did you ever have to do this to more than one server in your enterprise? I have, and it was always such a hassle to go from server instance to server instance, querying system objects, checking various configuration settings, and scouring through all sorts of places to ensure that your SQL Server instances were all compliant. That process has changed in SQL Server 2008.

Policy Management in SQL Server 2008

Yes, it's true. SQL Server 2008 introduces a new feature known as the Policy-Based Management. This framework allows you to define policies on a variety of objects and then either manually or automatically prevent changes based on said policies. Management is also very simple using SQL Server Management Studio (preferred), or you can write your own code to manage policies. But I am getting ahead of myself. Let's start at the beginning.

This management framework allows you to easily and proactively manage a variety of policies, ranging from security to metadata. It does this in two ways:

- It allows you to monitor changes to these policies, with options to manually check policies, check policies on schedule, check policies on change and log violations, or check policies on change and prevent the change if the policy is violated.

- It allows you to manage one or more SQL Server instances on a single server or across multiple servers.

Rather than waiting for something to go awry, you can set policies based on your server specifications and then have the framework proactively prevent changes based on these policies or inform you via policy logs when these policies are being violated. The ability to prevent certain changes depends on the type of feature, or facet, for which you are creating a policy. For example, if you want to ensure that xp_cmdshell is never turned on for any sever you are managing, you can create a policy and have it inform you when a change occurs or even have it check for changes on a schedule, but you cannot prevent it from being changed. The ability to prevent changes varies from facet to facet.

Policy-Based Management in SQL Server Management Studio

The practice of creating and enforcing policies is easily achieved using SQL Server Management Studio. Policy-Based Management is accessed primarily by the Policy Management node in Object Explorer, which can be found under the Management node of the SQL Server instance, as shown in Figure 1-1.

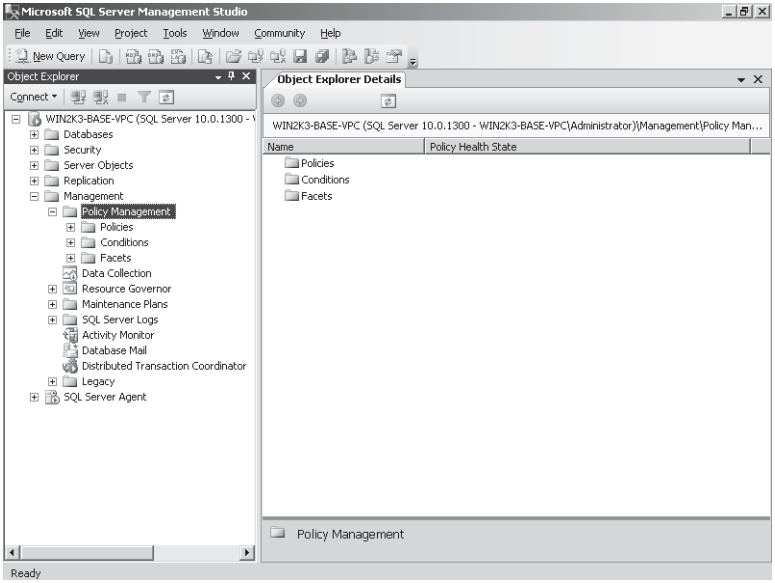


FIGURE 1-1 Policy Management in Object Explorer

Within this node of Object Explorer, you find the three base items of the framework: Policies, Conditions, and Facets. Although not shown as a node, Policy Category Management can also be accessed from here by right-clicking on the Policy Management node of Object Explorer and choosing Manage Categories. So what does each of the objects do to help you

implement policy-based management? Let's dig into each of them in more detail and discover how they are used.

Policy-Based Management Objects

Policy-Based Management uses five different objects to manage policies: facets, conditions, policies, targets, and categories.

Facets

Facets are the base units of this framework. Facets are types of objects, such as a Surface Area feature, server, logon, database, user, and so on. Each facet has a set of predefined properties against which conditions can be created.

As of the Community Technology Preview 6 (CTP6) release, there are a total of 47 facets, with a whopping 1,492 total properties. SQL Server Management Studio has a list of these facets under the Facets node in Objects Explorer (found under Management, Policy Management). Alas, if you want to see each list of properties, you need to open each facet's properties individually. If you want a quick list of all facets and properties, however, you can use the SQL Server Management Objects (SMO) to iterate through all available facets and properties, as shown here:

```
FacetInfoCollection fic = PolicyStore.Facets;
IEnumerable<FacetInfo> fic_sorted = from fic_i in fic
                                    orderby fic_i.DisplayName
                                    select fic_i;
Int32 pcount;
foreach (FacetInfo fi in fic_sorted)
{
    Console.WriteLine("FACET: " + fi.DisplayName);
    IEnumerable<PropertyInfo> fi_sorted =   from fi_i in fi.FacetProperties
                                            orderby fi_i.Name
                                            select fi_i;
    pcount = 0;
    foreach (PropertyInfo pi in fi_sorted)
    {
        if (pcount++ > 0)
            Console.Write(", " + pi.Name);
        else
            Console.Write(pi.Name);
    }
    Console.WriteLine();
    Console.ReadLine();
}
Console.WriteLine("---End of List---");
Console.ReadLine();
```

Facets by themselves cannot do anything in establishing policies. They can be used by conditions, however, to define what rules you want to create and against which servers, databases, or other objects the policies should check.

Conditions

A condition is an expression that defines the desired state of a facet. You express a condition by setting a facet property, a comparative operator, and a value. Each property condition's state is set according to its respective data type. For example, the Name property of the Stored Procedure facet is of type String and can have a condition operator of equal (=), not equal (!=), LIKE, NOT LIKE, IN, or NOT IN. Thus it can be compared with a string or a list of strings. The SQL Mail property of the Surface Area facet is of data type Boolean, and thus it has only the equality and inequality operators and can only be set to a value of true or false.

Note There is an advanced expression editor (the Advanced Edit dialog box) available if you need to create a specialized condition check. For example, you can check that the name of a table doesn't equal the schema name or that all tables have a primary key. The advanced expression editor allows a lot of flexibility, but when used in a condition, its respective policy can only be executed On Demand.

Both the field and expression value can be set using the advanced expression editor. In addition to providing a custom expression, it also provides an explanation of the available functions and a description of the facet properties. So if you are not sure what the property represents, you do not need to go to the facet and open it; you can simply click the ellipsis button (...) and examine the properties from there.

Furthermore, a condition can also only contain properties from a single facet type. For example, you can create a condition that states "SQL Mail is disabled and Database Mail is disabled" because both of these properties are part of the Surface Area facet. You cannot, however, create a condition that states "stored procedure names must begin with 'pr' and xp_cmdshell is disabled" because these two properties are part of two different facets (the Stored Procedure facet and Surface Area facet, respectively).

You can, however, create multiple conditions based on the same underlying facets. So you can create a condition that states "SQL Mail is disabled and Database Mail is disabled," and you can create a second condition that states "SQL Mail is disabled and Database Mail is enabled." Of course, you wouldn't want to have both policies on the same server because one of the policies will always be in violation.

SQL Server 2008 comes with an assortment of predefined conditions that you can immediately put into use. For example, one of my favorites is the condition named Auto Shrink Disabled, which can be used by a policy to ensure that databases do not enable the auto shrink option. Figure 1-2 shows this particular condition in the Open Condition window.

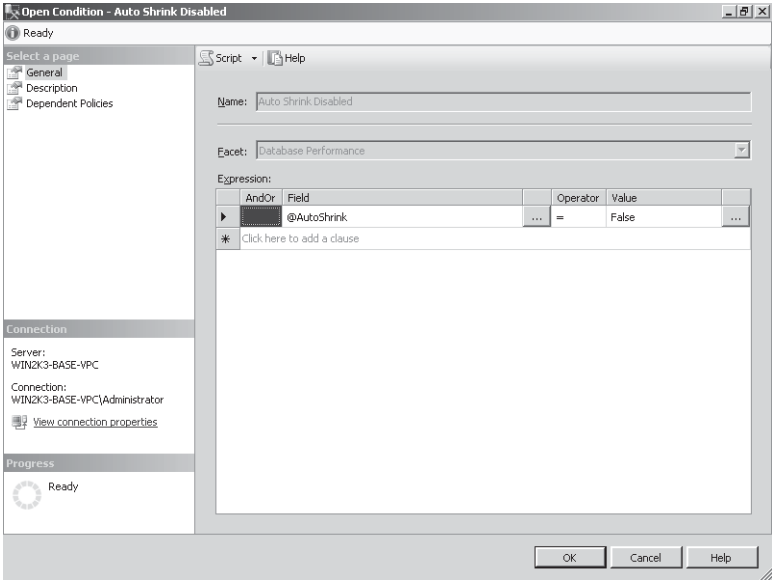


FIGURE 1-2 The Open Condition window

As I stated earlier in this section, you can also set multiple property states in a condition. Multiple conditions can each be set with an OR or AND clause, and they follow the standard order of operations. For example, Figure 1-3 shows an example of a new condition named Mail Features Disabled that states both SQL Mail and Database Mail are disabled.

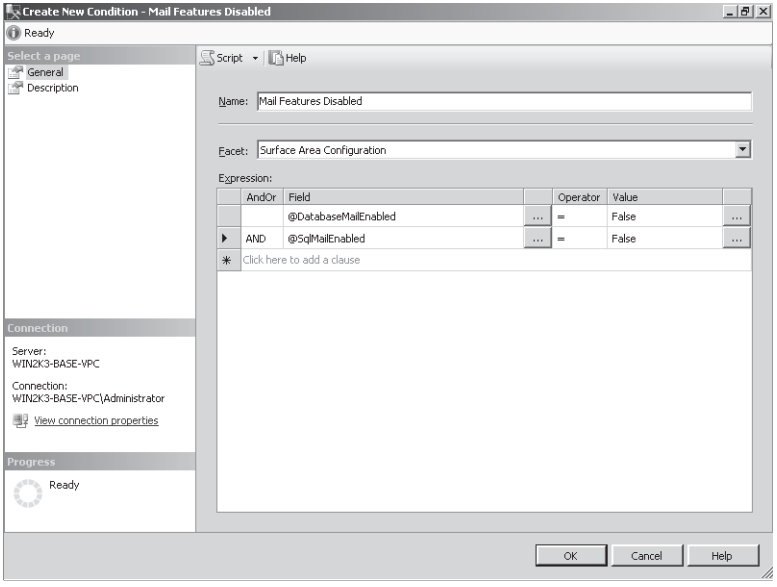


FIGURE 1-3 A new condition for disabled mail features

Policies

A policy is associated to a single condition and can be set to enforce or check the condition on one or more servers. The Execution Mode of the policy determines how a policy is enforced. Execution Mode can be set to one of four values:

- **On Demand** Do not check or enforce the policy. This is used to manually check policies.
- **On Schedule** Check the policy on a set schedule and log if policy is violated.
- **On Change - Log Only** Check the policy whenever a change occurs to the associated facet properties and log if the policy is violated.
- **On Change - Prevent** Check the policy whenever a change occurs to the associated facet properties and, if the policy is violated, prevent the change.

All policies can have an execution mode of On Demand or On Schedule. Only some, however, can be set to On Change - Log Only or On Change - Prevent. The execution mode setting of a policy is determined by the condition's underlying facet of the policy. Properties of certain facets can prevent attempted changes, whereas other facets can be checked on changes but only log when the policy is violated, and still others only checked on schedule.

Note The execution mode also determines whether the policy needs to be enabled or not. If the execution mode is set to On Demand, then the policy must be disabled. For all other execution modes, the policy can be enabled or disabled as needed. Keep in mind that if a policy is disabled, even if its execution mode is set to On Change - Prevent, it will not be checked and will not be automatically enforced.

How can you tell which facets support which execution modes? A quick query of the syspolicy_management_facets system view can give you the answer:

```
USE msdb;
GO

; WITH AutomatedPolicyExecutionMode (ModeId, ModeName)
AS
(   SELECT *
    FROM
        (VALUES (0, 'On Demand')
            , (1, 'On Change - Prevent')
            , (2, 'On Change - Log Only')
            , (4, 'On Schedule')
        ) AS EM(ModeId, ModeName)
 )
```

```
SELECT
    pmf.[management_facet_id] AS FacetID
    , pmf.[name] AS FacetName
    , APEM.[ModeName]
FROM syspolicy_management_facets AS pmf
    INNER JOIN AutomatedPolicyExecutionMode AS APEM
        ON pmf.[execution_mode] & APEM.[ModeId] = APEM.[ModeId]
ORDER BY pmf.[name], APEM.[ModeName]
```

This query will show you a list of facets and their supported execution modes. Abridged results are shown here:

```
FacetID  FacetName               ModeName
1        ApplicationRole         On Change - Log Only
1        ApplicationRole         On Change - Prevent
1        ApplicationRole         On Demand
1        ApplicationRole         On Schedule
2        AsymmetricKey           On Demand
2        AsymmetricKey           On Schedule
3        Audit                   On Demand
3        Audit                   On Schedule
4        BackupDevice            On Demand
4        BackupDevice            On Schedule
5        CryptographicProvider   On Demand
5        CryptographicProvider   On Schedule
6        Database                On Change - Log Only
6        Database                On Demand
6        Database                On Schedule
```

How the policy is enforced is only a part of the concept of policies. Figure 1-4 shows an example of a policy that checks the Mail Features Disabled condition but does it only in SQL Server 2005 or later.

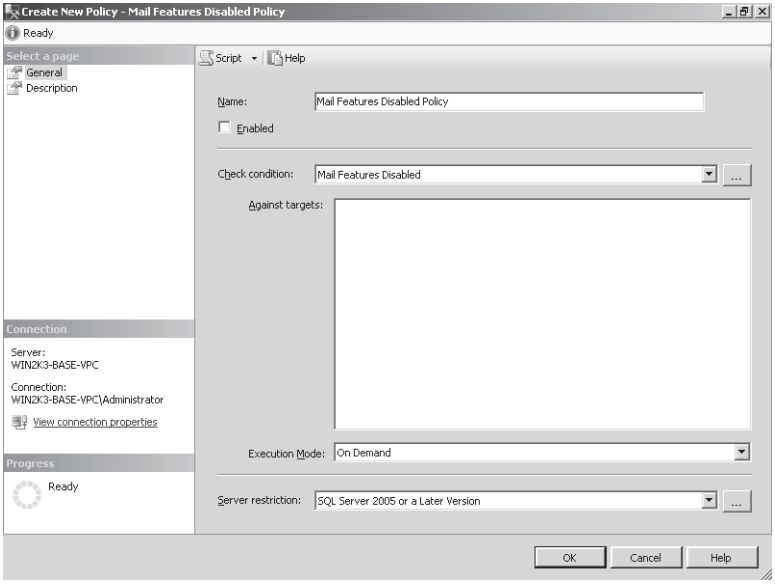


FIGURE 1-4 A new policy for Mail Features

Target Sets

Conditions are the basis for checks done by policies, but they can also be used to filter policies against target sets. A target set consists of one or more objects in the object hierarchy (i.e., server, database, and table) and conditions used to filter which of these objects the policy checks.

Target sets are broken into two categories. The first is for the server, which is used to filter which servers the policy performs its checks. The second is for the hierarchy of the database and its child objects (tables, views, stored procedures, and so on). For example, suppose you are implementing a policy for a condition that states Database Mail and SQL Mail are disabled. Such a policy wouldn't be applicable for SQL Server 2000, so you would want the policy to apply only to SQL Server 2005 or a later version.

The first step would be to create a condition for Database Mail and SQL Mail disabled (as shown in a previous example for the condition named Mail Features Disabled). Next you would create a condition, as shown in Figure 1-5, for the server's major version greater than or equal to 9 (SQL Server 2000 is version 8, 2005 is version 9, and 2008 is version 10). This condition named SQL Server 2005 Or A Later Version is actually created as a predefined condition on installation of SQL Server 2008.

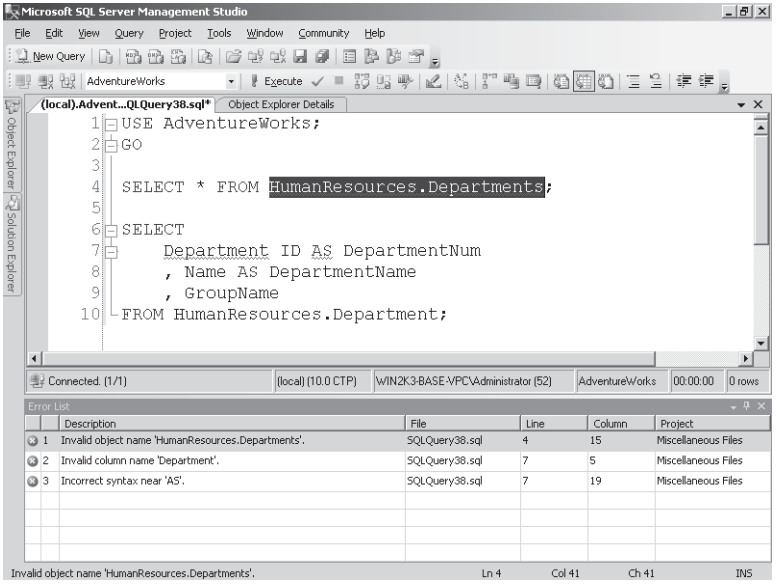


FIGURE 1-5 Condition for SQL Server version 9 (2005) or greater

Finally, you would create the policy that would check the Mail Features Disabled condition but additionally specify the SQL Server 2005 Or A Later Version condition in the Server Restriction drop-down list, as shown earlier in Figure 1-4.

Another way to use a condition to define a target set is by way of the Against section of the policy, which allows you to create the target set for the database hierarchy. For example, perhaps you want to enforce a naming convention for tables such that tables cannot start with the prefix "tbl." First you create a condition named Table Name on the Table facet that states: @Name NOT LIKE 'tbl%'. From here, you create a policy named Table Name Best Practice that checks the Table Name condition. Next, in the Against section, you specify that the check is only done against Non-System Tables in Online User Database (two more predefined conditions that come installed with SQL Server 2008). Now the check would only apply to non-system tables in online user databases. Figure 1-6 shows this policy and how you can choose the target condition for database.

You may have noticed that the policy that used the Mail Features Disabled condition didn't have any option available in the Against Targets section. The reason for this is that the Against Targets section applies only to objects lower than server in the hierarchy. The Server Restriction option applies for servers themselves, so if your policy is based on a condition that is at the server level (server, server performance, server configuration, and so on), it will not have any options for lower-level target sets (such as databases, tables, columns, and so on).

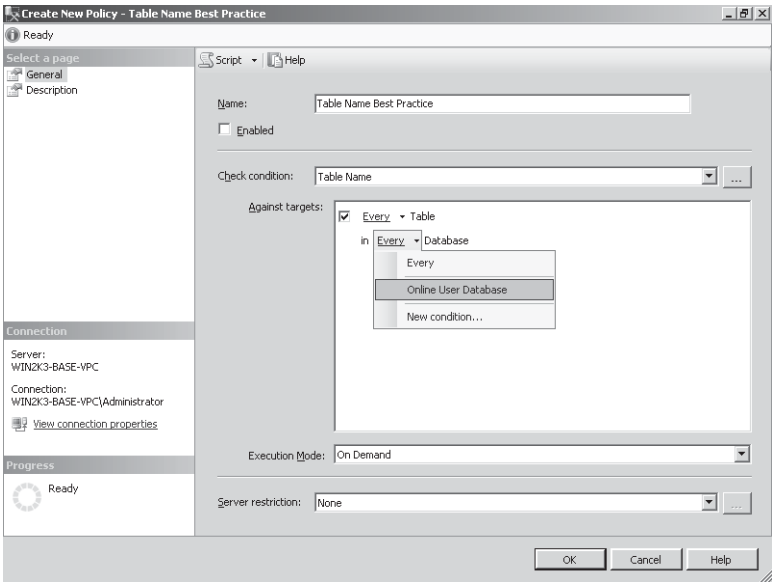


FIGURE 1-6 Policy with target filter

By using targets, however, in conjunction with condition checks, you can use different types of underlying facets in a single policy by using a condition for the check and for each level in the object hierarchy.

Policy Categories

Policy categories are used to group policies and can be used to force policy checks or allow database owners to optionally subscribe to a set of policies. A policy can belong to only one policy category, either user-defined or the Default category. Policy categories can be created on the fly when defining a policy and can be further managed through the Manage Policy Categories dialog box. In this dialog box, one can also determine if category subscriptions at the database level are mandated or optional, as shown in Figure 1-7.

Yes, you read that correctly—mandated. You can create and enable a set of policies, group them in one or more policy categories, and then force all databases to subscribe to these policies.

If you don't assign a policy to a policy category, it is placed in the Default policy category, which always mandates a subscription from databases. Unlike other policy categories, the Default policy category cannot be changed to optionally allow subscriptions. So if you put a policy in this policy category and the policy is enabled and enforced (On Change - Prevent), then all databases will have to comply. If you want the ability to optionally allow subscriptions to the policy category, you must add the policy to a policy category other than Default

and then use the Manage Policy Categories dialog box to set the policy category mandate subscription option as false (clear the check box).

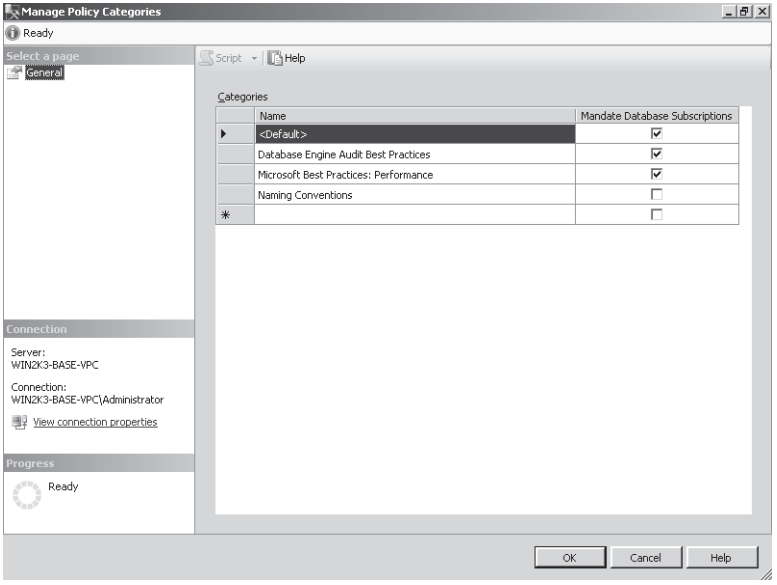


FIGURE 1-7 The Manage Policy Categories dialog box

Policy Checking and Preventing

As mentioned earlier in this chapter, not all policies can be set to prevent changes when a policy is violated, but you can check policies and log violations, both on change or on schedule. But what happens when you do a manual check and you find a policy is being violated? What is the process to remedy the violation on the server, database, and so on?

Let's revisit an example from earlier in the chapter. You create a policy that states "SQL Mail and Database Mail should both be disabled." You set its Execution Mode to On Demand and leave the policy disabled. Now, how do you go about checking the policy?

First we are going to set the server configuration so that it will violate the policy by running the following Transact-SQL (T-SQL) code.

```
--Run this first to see advanced options
EXEC sp_configure 'show advanced options', 1
RECONFIGURE WITH OVERRIDE

--Run this second to change the mail configuration
EXEC sp_configure 'Database Mail XPs', 1
EXEC sp_configure 'SQL Mail XPs', 0
RECONFIGURE WITH OVERRIDE
```

Now your server will fail the policy check. Next, we evaluate the policy by right-clicking on the SQL Server instance in Object Explorer and choosing Policies, then View, as shown in Figure 1-8.

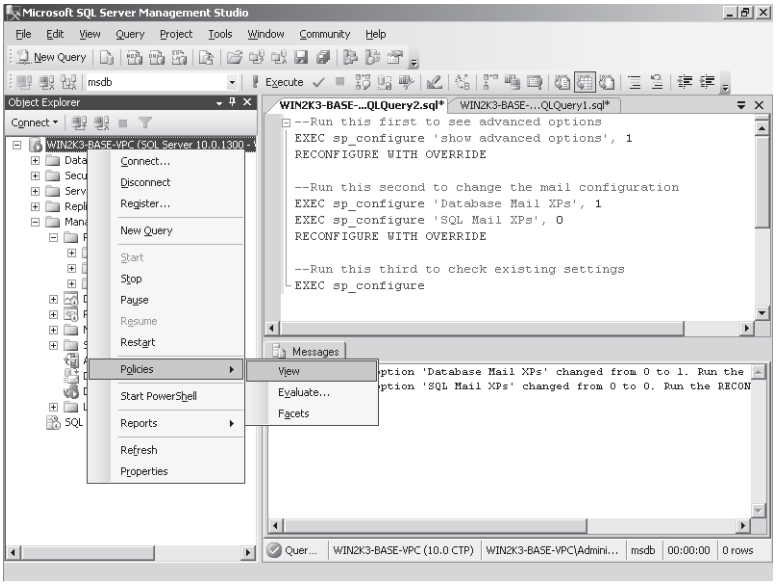


FIGURE 1-8 Opening the View Policies dialog box

This will bring up the View Policies dialog box, shown in Figure 1-9, where you can view information about the policies, including whether the policy is enabled (Effective), the policy's category, the policy's last execution, and comments. Here you can also click to see a history of the policy and to evaluate the policy.

Note If you want to see a history of all policies, you can right-click the Policies node in Object Explorer and, from the context menu, choose View History.

As shown in Figure 1-10, clicking Evaluate reveals that the server is violating the policy (as expected because we purposefully ran script to violate the policy earlier in this section).

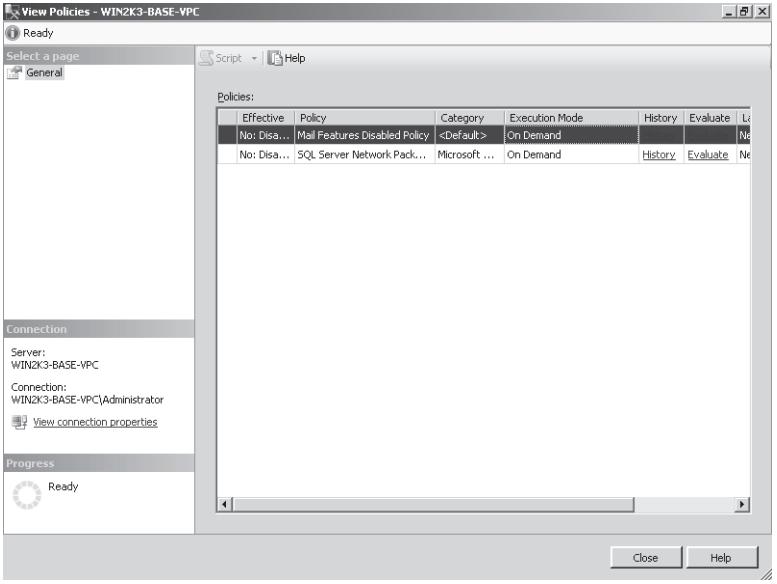


FIGURE 1-9 View Policies dialog box

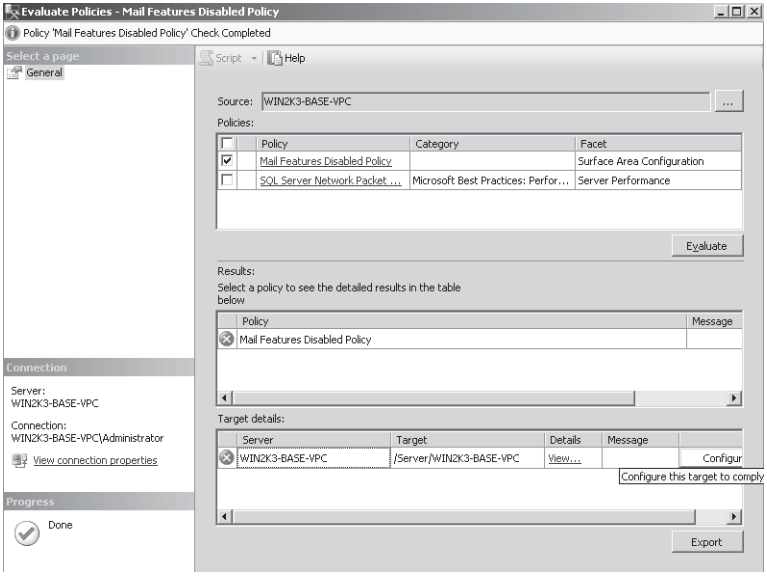


FIGURE 1-10 Evaluate Policies dialog box for the Mail Features Disabled Policy

Clicking the Evaluate button will cause the policy to be checked again and will result in the same thing—a policy that is in violation. But clicking the Configure button will simply fix the problem, as shown in Figure 1-11.

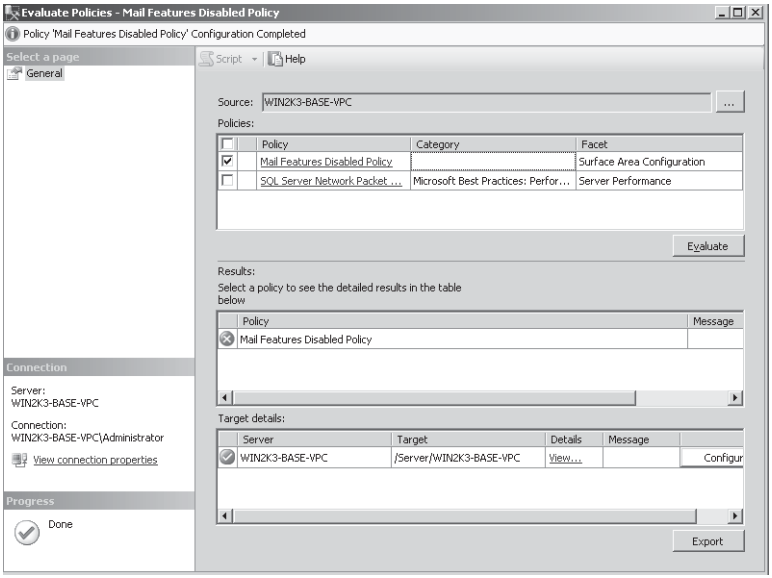


FIGURE 1-11 Resolving a policy violation

Indeed, a single click and you can remedy policy violations on the server. It makes the appropriate changes to the configuration and reruns the policy check, resulting in a policy that is now compliant.

Note You can also view and test (check) individual policies by right-clicking the policy located under the Policy node in Object Explorer, or you can view policies for other objects in Object Explorer, such as a database or a table.

Policy-Based Management in Practice

At this point, we will run through a simple example of using the framework to enforce naming conventions on our tables, stored procedures, and functions. Here is a list of conditions you will need to create:

Condition Name	Facet	Expression
Stored Procedure Name	Stored Procedure	@Name NOT LIKE 'sp[_]%'
Table Name	Table	@Name NOT LIKE 'tbl%'
Function Name	User Defined Function	@Name LIKE 'fn%'

The next step is to create three corresponding policies that are all part of the same category named Naming Conventions. All policies should use the default settings for Against Targets, Server Restriction, and Enabled, and the Execution Mode should be set to On Change - Prevent.

Policy Name	Condition
Stored Procedure Name Policy	Stored Procedure Name
Table Name Policy	Table Name
Function Name Policy	Function Name

Figure 1-12 shows an example of the Stored Procedure Name Policy and its appropriate settings.

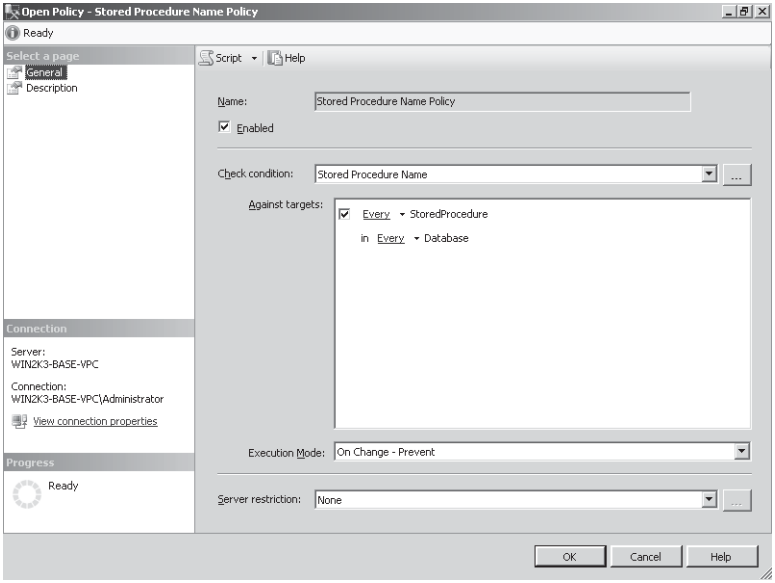


FIGURE 1-12 Stored Procedure Name Policy dialog box

By default, all new categories are set to mandate subscriptions, so using the *AdventureWorksLT* example database, we can try to create the various objects, as shown below.

```
USE AdventureWorksLT
GO

CREATE PROCEDURE sp_test
AS
    SELECT 1 As [one]
GO
```

```
CREATE TABLE tblTest
(
    tbltestID int NOT NULL,
    Description varchar(100) NULL
)
GO

CREATE FUNCTION fTest (@i INT)
RETURNS INT
AS
BEGIN
    RETURN @i * 2
END
GO
```

Running this script will result in the following:

```
Policy 'Stored Procedure Name Policy' has been violated by '/Server/(local)/Database/
AdventureWorksLT/StoredProcedure/dbo.sp__test'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
Msg 3609, Level 16, State 1, Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.

Policy 'Table Name Policy' has been violated by '/Server/(local)/Database/
AdventureWorksLT/Table/dbo.tblTest'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
Msg 3609, Level 16, State 1, Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.

Policy 'Function Name Policy' has been violated by '/Server/(local)/Database/
AdventureWorksLT/UserDefinedFunction/dbo.fTest'.
This transaction will be rolled back.
Policy description: ''
Additional help: '' : ''.
Msg 3609, Level 16, State 1, Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.
```

You will notice that there is additional information such as Policy Description, which is simply an empty string. You can include this additional description to add information in the policy. Figure 1-13 shows an example of setting a description, help text, and URL for the Stored Procedure Name Policy.

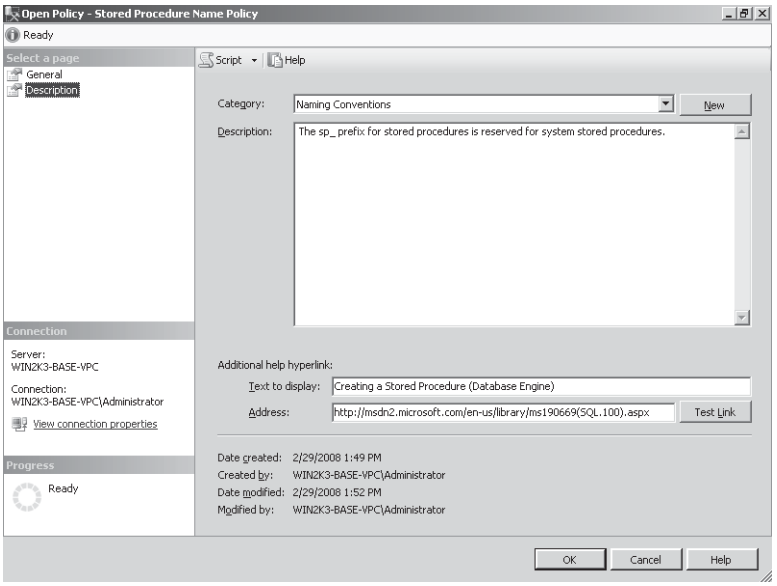


FIGURE 1-13 Policy description settings

Using the information shown in Figure 1-13, change the Stored Procedure Name Policy description information, and try to create the stored procedure again. The results will now show (with changes shown in italics):

```
Policy 'Stored Procedure Name Policy' has been violated by '/Server/(local)/Database/
AdventureWorksLT/StoredProcedure/dbo.sp__test'.
This transaction will be rolled back.
Policy description: 'The sp_ prefix for stored procedures is reserved for system
stored procedures.'
Additional help: 'Creating a Stored Procedure (Database Engine)' : 'http://msdn2.
microsoft.com/en-us/library/ms190669(SQL.100).aspx'.
Msg 3609, Level 16, State 1, Procedure sp_syspolicy_dispatch_event, Line 50
The transaction ended in the trigger. The batch has been aborted.
```

Auditing SQL Server

Over the years I've seen and heard of a variety of solutions used to audit actions performed in SQL Server. Prior to SQL Server 2005, it was difficult to easily and efficiently audit particular actions, such as when someone changed an object definition or when someone selected data from a table or view. How did you know if someone added a column to a table or changed a view's definition or ran a select statement against a table?

C2 Audit Mode

One option was to use the C2 audit mode for SQL Server, available since SQL Server 2000. However, a C2 audit captures a lot of audit events, and that could mean many megabytes per minute on the hard disk of your default data directory. This could have some performance implications for the server.

C2 audit mode is black and white as far as what is audited, so you are either auditing everything (C2 audit mode on) or nothing (C2 audit mode off). Switching between on and off, however, requires a restart of the SQL Server instance.

To view the audit data, you could use SQL Profiler and load in the trace file. From there you could push the trace file data into a table. Another option is to use the fn_trace_gettable system function to view the data directly in SQL Server Management Studio (SSMS).

Other Audit Techniques

If C2 audit mode is more than you need, there are other creative techniques used to audit a more specific set of actions. For example, you could "audit" selects against a table if you used stored procedures as the basis for all select statements. You could audit metadata changes by scripting the objects on a regular basis and comparing the versions. Data manipulation language (DML) triggers can be used to audit changes to data. And although you can usually find a solution, implementation is sometimes cumbersome, and each type of audit requires a different type of solution.

SQL Server 2005 then introduced data definition language (DDL) triggers. This new feature made auditing somewhat more manageable, allowing you to capture more efficiently changes to metadata. I have had several clients benefit from even the most primitive of metadata audits using DDL triggers. This new ability still only remedied one of the areas of auditing.

Many of you may be thinking, "I could use SQL Profiler and capture many of these events." And it's true—you could run a trace to capture audit information. Traces, however, have to be started every time the server restarts, and there are other limitations, especially when filtering. For example, if you want to audit inserts into SalesOrder table for users in the Sales role and you also want to audit inserts into the Customer table for users in the Marketing roles, you would not be able to do so in a single trace. Your best bet would be to use multiple traces or to trace inserts for both roles against both tables.

Auditing in SQL Server 2008

SQL Server 2008 brings auditing to a new level, with a robust auditing feature set. There are 81 securable types grouped into 22 classes. The securable types include items such as the server, logins, certificates, tables, indexes, keys, roles, schemas, triggers, endpoints, and

message types. Each of these securable types can have a variety of actions audited. For example, you can audit when someone changes the definition of, selects from, inserts into, deletes from, or updates a table.

The Audit

The first step for auditing is to create an Audit object. An Audit object is a container for audit specifications, both at the server and database levels. It is associated with a single server instance (audits do not work against multiple servers) and can record audit data to one of the following locations:

- The Application Event Log
- The Security Event Log
- The File System (one or more files on a local drive or network share)

Note The service account for the instance of SQL Server that is implementing an audit needs to have enough privileges to do its job. So if writing to the file system, the service account must be able to read, write, and modify. If writing to the Security Event Log, the service account needs the Generate Security Audits user right (which is by default only given to Local Service and Network Service), and the Windows *Audit* object needs to be configured to allow access, which is done through auditpol.exe in Vista/W2K8 and secpol.exe on earlier versions of Windows.

There are two ways to go about creating an audit. First, using Object Explorer in SSMS, navigate to the <Server_Instance>/Security/Audits node. Right-click the node, and choose New Audit. That will open the Create Audit dialog box, as shown in Figure 1-14.

This example is using a one second queue delay, meaning the audit will write its data asynchronously to the destination within one second of the event. Choosing a value of 0 for the queue delay means processing is done synchronously and the transaction will block until the data is written to the destination. This example also shows the audit data being sent to a file location. Normally you would choose something other than your system drive, such as a drive on a separate set of spindles or perhaps even a network share. You can also specify the size of the audit files (and optionally reserve space for it) and the number of rollover files.

The Shut Down Server On Audit Log Failure option does exactly as it implies—if the audit fails to work, the server instance shuts down. But what does it mean for an audit to fail? Some might think that this means when a failure event is recorded, such as a login failure, it causes the server to shut down. This is not true. Audit failure means the audit cannot record audit event data. For example, if the above audit was created and enabled but there was no directory C:\Audit\Security, then the audit would fail, and the server instance would shut down. You can restart the service, however, because the audit will be disabled because it

failed to start. My suggestion is to only turn on this option after you've first tested the audit with this option off.

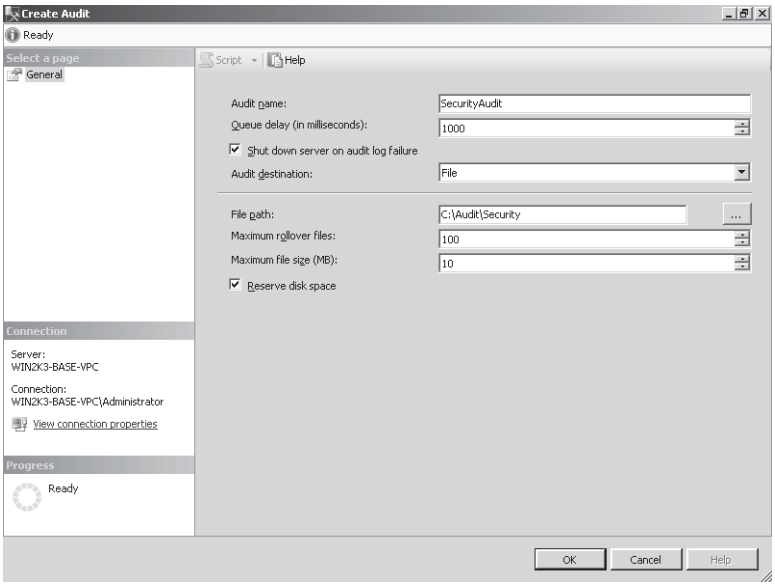


FIGURE 1-14 The Create Audit dialog box

Now for all of you who prefer to be able to script such changes, here is the equivalent action as done in T-SQL.

```
USE [master]
GO

CREATE SERVER AUDIT [SecurityAudit]
TO FILE
(    FILEPATH = N'C:\Audit\Login\'
    ,MAXSIZE = 10 MB
    ,MAX_ROLLOVER_FILES = 100
    ,RESERVE_DISK_SPACE = ON
)
WITH
(    QUEUE_DELAY = 1000
    ,ON_FAILURE = SHUTDOWN
    ,AUDIT_GUID = 'ab562f41-f4ad-af4c-28a0-feb77b546d89'
)
GO
```

All of the options that were available in the graphical user interface (GUI) dialog box are available using T-SQL. Using T-SQL, however, you have more flexible settings. Sure, either method allows you to specify up to 2,147,483,647 rollover files, but only using T-SQL can you

set the maximum file size up to 16,777,215 terabytes. If you happen to have more disk space available, you can simply specify UNLIMITED (equivalent to selecting 0 when using the dialog box). And only using T-SQL can you set the globally unique identifier (GUID) for the server audit. However, you cannot use a variable to do so, for the value must be static (unless you use dynamic SQL, of course). For example, the following T-SQL would raise an exception.

```
USE [master]
GO

DECLARE @guid UNIQUEIDENTIFIER = NEWID()

--This will fail
CREATE SERVER AUDIT [SecurityAudit]
TO FILE
(    FILEPATH = N'C:\Audit\Login\'
    ,MAXSIZE = 10 MB
    ,MAX_ROLLOVER_FILES = 100
    ,RESERVE_DISK_SPACE = ON
)
WITH
(    QUEUE_DELAY = 1000
    ,ON_FAILURE = SHUTDOWN
    ,AUDIT_GUID = @guid
)
GO
```

Note Why specify a GUID for an audit? To allow failover scenarios where you have database audit specifications and the database may fail over from one server to another and you want the audit to fail over as well. You define an audit on each server with the same GUID because the audit specifications link to audits by GUID.

Finally, you may be asking yourself what happens if you do reach the maximum file size for the maximum number of rollover files. It starts to roll over on the existing file(s). So if you have a maximum file size of 100 MB and a maximum of 10 rollover files, then you will have at most 1000 MB of audit data.

Audit Specifications

An audit is not very useful until it has audit specifications. These specifications determine which actions the audit records. An audit can have one server audit specification and one database audit specification per database.

Server Audit Specifications

Regardless of what action is being audited, these specifications audit at the server instance level and, therefore, exist only in the master database. Server audit specifications use server-level audit action groups to determine what actions are audited. These groups contain one or more actions that are audited at the server level. For example, the TRACE_CHANGE_GROUP is a container for five trace-related actions: starting a trace, stopping a trace, altering a trace, enabling a C2 trace audit, and disabling a C2 trace audit. The SCHEMA_OBJECT_ACCESS_GROUP, however, has 15 actionable events that can be audited.

Following the previous example, let's add a server audit specification that audits actions for failed logins and for changes related to login passwords, which includes resetting the password, changing a password, unlocking an account, and so on.

```
USE [master]
GO

CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpecForLogin]
    FOR SERVER AUDIT [SecurityAudit]
    ADD (FAILED_LOGIN_GROUP),
    ADD (LOGIN_CHANGE_PASSWORD_GROUP)
    WITH (STATE=ON)
GO
```

You will first notice that the server audit specification is being created in the context of the master database because it must be created in the master database. If the database context was otherwise, the statement would fail. When creating a server audit specification, you need to associate it with a server audit, and then you can optionally begin to specify the server-level audit action groups and its initial state, as shown in this example above.

When making changes to a specification, you must first disable it. This means that when using SSMS, you need to clear the Enabled check box to save the changes. Then, using Object Explorer, you can right-click the specification you just changed and choose to re-enable it from the context menu. Using T-SQL for such changes means you first alter the specification to disable it and then add the action groups and enable it using a second alter statement.

This next example shows how to add another action group using T-SQL, first by changing the state of the server audit specification to Off, and then by adding the action group and changing the state back to On.

```
USE [master]
GO

ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecForLogin]
    WITH (STATE=OFF)
```

```
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecForLogin]
    ADD (SUCCESSFUL_LOGIN_GROUP)
    WITH (STATE=ON)
GO
```

What if you need to drop one of the audited action groups? The ALTER SERVER AUDIT SPECIFICATION also has a DROP clause, which is used to remove action groups from the specification, as shown here.

```
USE [master]
GO

ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecForLogin]
    WITH (STATE=OFF)

ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecForLogin]
    DROP (SUCCESSFUL_LOGIN_GROUP)
    WITH (STATE=ON)
GO
```

One final note: Server audit specifications use server-level action groups, which are all encompassing on the server instance. For example, creating a server audit specification using the action group SCHEMA_OBJECT_ACCESS_GROUP will audit all SELECT, INSERT, UPDATE, DELETE, EXECUTE, RECEIVE, REFERENCES, and VIEW CHANGETRACKING for all objects in all databases. If your auditing needs are more specific, then you can use database audit specifications.

Database Audit Specifications

In some scenarios, you may not need to audit every database. You might want to audit only certain objects in a particular database. Database audit specifications allow you to audit more specific events in a particular database. Unlike server audit specifications, database audit specifications can audit both action groups and actions. They can also audit actions on specific database objects.

The following example continues from the previous examples, adding auditing for a variety of security-related actions.

```
USE [AdventureWorksLT]
GO

CREATE DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_Access]
FOR SERVER AUDIT [SecurityAudit]
ADD (DATABASE_ROLE_MEMBER_CHANGE_GROUP),
```

```
ADD (DATABASE_PERMISSION_CHANGE_GROUP),
ADD (DATABASE_OBJECT_PERMISSION_CHANGE_GROUP),
ADD (SCHEMA_OBJECT_PERMISSION_CHANGE_GROUP),
ADD (DATABASE_PRINCIPAL_IMPERSONATION_GROUP),
ADD (DATABASE_PRINCIPAL_CHANGE_GROUP),
ADD (DATABASE_OWNERSHIP_CHANGE_GROUP),
ADD (DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP),
ADD (SCHEMA_OBJECT_OWNERSHIP_CHANGE_GROUP)
WITH (STATE=ON)
GO
```

But what if you wanted to audit only when someone in the Marketing role issues a SELECT statement against the Employees table? Instead of simply adding the action group, you can also add a specific action, such as SELECT, but you must also specify a securable (class or object) and a principal (user or role). This next example shows how to add an audit specification at the database level that will record whenever someone in the Marketing role selects data from the Employees table in the HumanResources schema.

```
USE [SomeOtherDatabase]
GO

CREATE DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_DataAccess]
FOR SERVER AUDIT [SecurityAudit]
ADD (SELECT ON [HumanResources].[Employees] BY [Marketing])
WITH (STATE=ON)
GO
```

After creating this specification, we realize that we need a more general audit such that we know when anyone selects, inserts, updates, or deletes data from any object in the HumanResources schema. We can drop the original specification for SELECT and add a more general one that audits selects, inserts, updates, and deletes for all (applicable) objects in the HumanResources schema.

```
USE [SomeOtherDatabase]
GO

ALTER DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_DataAccess]
WITH (STATE=OFF)

ALTER DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_DataAccess]
DROP (SELECT ON [HumanResources].[Employees] BY [Marketing])
ADD (SELECT, INSERT, UPDATE, DELETE ON Schema::[HumanResources] BY [public])
WITH (STATE=ON)
GO
```

Note the syntax for a class (as opposed to a specific object), such as a schema, requires a prefix of the class name and a double colon, such as Schema::[SalesLT]. Because the database is also an object, you can also make this audit specification apply to all (applicable) objects in the database, as follows.

```
USE [SomeOtherDatabase]
GO

ALTER DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_DataAccess]
WITH (STATE=OFF)

ALTER DATABASE AUDIT SPECIFICATION [DatabaseAuditSpec_DataAccess]
DROP (SELECT, INSERT, UPDATE, DELETE ON Schema::[HumanResources] BY [public])
ADD (SELECT, INSERT, UPDATE, DELETE ON Database::[AdventureWorksLT] BY [public])
WITH (STATE=ON)
GO
```

Audit Results

I just read a post over at Microsoft Developer Network (MSDN) forums about using audit to track changes to data. That is not what auditing does. Auditing is used to tell you what actions have occurred and not what data has changed in the underlying tables. To see this information, you can use SSMS and view the audit logs, you can view the Windows Application or Security Event logs (both are also viewable from SSMS), or, if you stored the audit in a file location, you can query the audit log files directly.

To view using SSMS, navigate to the <Server_Instance>/Security/Audits node, right-click on the audit whose log you want to view, and choose View Audit Logs. This will open the Log File Viewer dialog box, as shown in Figure 1-15. From here, you can browse all the server audits as well as the Windows Application and Security Event logs.

Using T-SQL, you can use the system table–valued function fn_get_audit_file to query the audit data. Not only can this function query individual files, but it can also query all files in a specific directory. For example, the audit example earlier in this section used the file location C:\Audit\Security. If other audits had used this same directory, then all audits would be returned via this query.

```
SELECT *
FROM sys.fn_get_audit_file
('C:\Audit\Security\*', DEFAULT, DEFAULT);
```

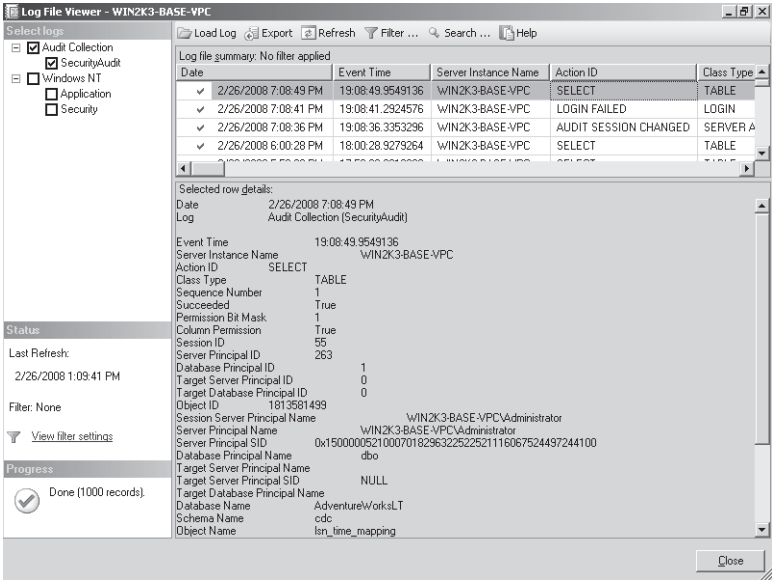


FIGURE 1-15 Log File Viewer dialog box

But if you only wanted audits with the GUID {AB562F41-F4AD-AF4C-28A0-FEB77B546D89}, then you could run the following query to get those results.

```
SELECT *
FROM sys.fn_get_audit_file
('C:\Audit\Security\SecurityAudit_AB562F41-F4AD-AF4C-28A0-FEB77B546D89*', DEFAULT,
DEFAULT);
```

The second and third arguments for this function are a starting file and a starting offset within the file.

```
SELECT *
FROM sys.fn_get_audit_file
('C:\Audit\Security\*',
'C:\Audit\Security\SecurityAudit_AB562F41-F4AD-AF4C-28A0-FEB77B546D89_0_
128484368541270000.sqlaudit',
1024);
```

The fn_get_audit_file function returns a table that contains information about the event, including the date and time of the event, the objects and principals involved, the statement that was executed, miscellaneous additional data (if any), and information about the audit

file and offset within that file for the recorded event. SQL Server Books Online contains a detailed description of the return columns. Because of formatting, I will show one more query that returns results as XML, which is more easily displayed in this book format.

```
SELECT
    [event_time] AS [EventTime]
  , [action_id] AS [ActionId]
  , [succeeded] AS [Succeeded]
  , [session_server_principal_name] AS [ServerPrincipalName]
  , [database_principal_name] AS [DatabasePrincipalName]
  , [server_instance_name] AS [ServerInstanceName]
  , [database_name] AS [DatabaseName]
  , [schema_name] AS [SchemaName]
  , [object_name] AS [ObjectName]
  , [statement] AS [Statement]
  , LEFT([file_name], 20) + '...' AS [AuditFileName]
  , [audit_file_offset] AS [AuditFileOffset]
FROM sys.fn_get_audit_file ('C:\Audit\Security\*',DEFAULT, DEFAULT)AS [AuditData]
WHERE action_id = 'SL'
  AND [audit_file_offset] <= 2048
FOR XML PATH('AuditData')
```

Executing this query results in the following (based on my current audit definitions):

```
<AuditData>
  <EventTime>2008-02-26T17:18:09.7768064</EventTime>
  <ActionId>SL  </ActionId>
  <Succeeded>1</Succeeded>
  <ServerPrincipalName>WIN2K3-BASE-VPC\Administrator</ServerPrincipalName>
  <DatabasePrincipalName>dbo</DatabasePrincipalName>
  <ServerInstanceName>WIN2K3-BASE-VPC</ServerInstanceName>
  <DatabaseName>AdventureWorksLT</DatabaseName>
  <SchemaName>SalesLT</SchemaName>
  <ObjectName>Product</ObjectName>
  <Statement>SELECT * FROM SalesLT.Product</Statement>
  <AuditFileName>
    C:\Audit\Security\SecurityAudit_AB562F41-F4AD-AF4C-28A0-FEB77B546D89...
  </AuditFileName>
  <AuditFileOffset>1024</AuditFileOffset>
</AuditData>
<AuditData>
  <EventTime>2008-02-26T17:19:04.1549984</EventTime>
  <ActionId>SL  </ActionId>
  <Succeeded>1</Succeeded>
  <ServerPrincipalName>WIN2K3-BASE-VPC\Administrator</ServerPrincipalName>
  <DatabasePrincipalName>dbo</DatabasePrincipalName>
  <ServerInstanceName>WIN2K3-BASE-VPC</ServerInstanceName>
```

```
  <DatabaseName>AdventureWorksLT</DatabaseName>
  <SchemaName>SalesLT</SchemaName>
  <ObjectName>Product</ObjectName>
  <Statement>UPDATE [SalesLT].[Product] set [Color] = @1  WHERE [ProductID]=@2</
Statement>
  <AuditFileName>
    C:\Audit\Security\SecurityAudit_AB562F41-F4AD-AF4C-28A0-FEB77B546D89...
  </AuditFileName>
  <AuditFileOffset>2048</AuditFileOffset>
</AuditData>
```

Note It wouldn't take much to use SQL Server Integration Services in conjunction with the new auditing feature set to create and populate an audit data warehouse. Because the fn_get_audit_file function can specify a point in time for showing data, some SQL Server Integration Services (SSIS) package could get the last known file and offset and use that as a seed to get any new audit data.

Bonus Query

While working with audits, I ended up writing this query that will return all audit specification actions and groups, in their hierarchical order from the server level down to the object level.

```
;WITH AudAct AS
(
    SELECT
        CAST(class_desc COLLATE SQL_Latin1_General_CP1_CI_AS
             + '.' + [name] + '/' as varchar(8000))
             COLLATE SQL_Latin1_General_CP1_CI_AS  as [path],
        1 as [level],
        *
    FROM sys.dm_audit_actions
    WHERE configuration_level = 'Group'
      AND name = containing_group_name
      AND covering_parent_action_name is null
      OR (containing_group_name is null)

    UNION ALL

    SELECT CAST(AA1.[path] +
                ISNULL(AA2.covering_action_name + '.', '') +
                AA2.class_desc COLLATE SQL_Latin1_General_CP1_CI_AS + '.' +
                AA2.[name] + '/'
              as varchar(8000)) COLLATE SQL_Latin1_General_CP1_CI_AS  as [path],
           AA1.[level] + 1,
           AA2.*
    FROM AudAct AS AA1
        INNER JOIN sys.dm_audit_actions AS AA2
```

```
            ON
            (    AA2.parent_class_desc = AA1.class_desc
                AND
                (    AA2.covering_parent_action_name = aa1.name
                     OR
                    (AA2.covering_parent_action_name is null
                     AND AA2.covering_action_name = aa1.name)
                )
            )
            OR
            (  AA2.covering_parent_action_name is null
               AND AA2.covering_action_name IS null
               AND AA2.parent_class_desc is null
               AND AA2.containing_group_name = AA1.name
               AND AA2.configuration_level IS NULL
            )
)
SELECT * FROM AudAct
ORDER By [path]
```

Transparent Data Encryption

Prior to SQL Server 2005, most folks would either encrypt the data before sending it to SQL Server or would use extended stored procedures to do the encryption as needed. The advent of SQL Server 2005 introduced native encryption capabilities that were previously only available in external code.

What Is Transparent Data Encryption?

Unlike the encryption libraries and capabilities that were introduced in SQL Server 2005, which targeted encrypting columns and encryption on a user or role basis, transparent data encryption (TDE) is focused on encrypting your data without you having to do any additional programming work and with very little administrative setup. As a matter of fact, with very little effort, you could implement TDE and not require any programmatic changes.

TDE can encrypt all data in the database, such that the physical database data and transaction log files are encrypted. It does this by encrypting the data as it is written to disk and then decrypting it as it is read from disk.

What TDE Is Not

- TDE cannot encrypt or decrypt individual pieces of data.
- It cannot be used to restrict access to data.
- It cannot be used to secure data from individual users or roles.

- It is not usable as a row-level security implementation.
- It cannot prevent application users from accessing secure data.
- It is not aware of application security, Active Directory security, or other security mechanisms.

You may be asking yourself why I am making such a big deal about what TDE is not, and you would be right to do so. But people have already inquired about its usefulness for such situations as I just mentioned, and I want to be clear that it is meant to encrypt the physical storage of the data and nothing more.

Why Use TDE

Imagine you are a developer working remotely on a project with a client, and, for one reason or another, you cannot connect to the development database over a virtual private network (VPN) either due to technical difficulties or because the company does not allow vendors VPN access because of company security policies. The development database is large enough (tens of gigabytes or more) that e-mail, File Transfer Protocol (FTP) (secure, of course), Web Distributed Authoring and Versioning (WebDav), and so on are not feasible solutions for getting a local copy of the data. And more likely, the company also doesn't allow such access from external sources, again because of security policies. This company runs a tight ship, with multiple firewalls and layers of security to keep the bad people out. So how are you going to get the data so you can work?

So this company has you, the developer, sign a myriad of non-disclosure agreements (NDAs) and other legal documents to ensure the data that would be offsite is secure. Company officials require you to have a firewall, strong passwords, antivirus software, and spyware protection. They take measure after measure to ensure you are a safe person to work with, even doing a background check. And then, after they finally know they can trust you, they put a database backup or possibly a copy of a detached database on a portable hard drive and then they call some express mail service to come and pick up this hard drive and bring it to you.

Up until the moment the company called for the package pickup, it was doing everything in its power to keep the data secure: limiting VPN access only to employees, not allowing FTP or WebDav access, doing background checks, and so on. And then the entire security process broke down and was made completely and utterly vulnerable. Sure, I've never known anyone personally who had this data not arrive or used for ill purposes, but I've read reports of people who have lost data and been left exposed.

It doesn't even have to be such a complex situation. It could be something as simple as the offsite backup storage getting robbed.

When to Use TDE

For many scenarios, TDE is going to be an efficient and easy-to-implement encryption solution. It gives you instant protection of your data (including backups), especially data that will be sent or stored offsite.

Unless you only need to encrypt very little data and that encryption can be done in the middle tier, you will most likely see a benefit from using TDE. If you are storing a lot of highly sensitive information, including personal data, credit card information, Social Security numbers, private transactions, and the like, then you most definitely want to use TDE to secure your data.

Tip There are other factors besides how much of your data needs to be secured that would go into making the decision to use TDE or to simply encrypt individual data columns. Other factors that will also affect your decision include whether your physical data or backups are secure, how much of your data actually fits in cache, and how frequently data is modified.

How Does TDE Help?

If the data on the hard drive is encrypted (backup or detached database) and it requires the use of a certificate to make use of the physical data on the portable hard drive that the company mailed to you and, very importantly, the certificate and its credentials were not sent along with the hard drive, then the data on the hard drive is secure, even if it falls into the wrong hands. The same holds true for the offsite backup storage.

In addition to the encrypted data, you also need a certificate file, a private key file, and the credentials for that private key, all of which could be sent independently of each other or stored independently of the backups, thus keeping the security level as high as possible, which was the original intent for the data.

Before I dig into the implementation, I want to discuss a little more about the mechanism involved in transparently encrypting data at the database level.

How Does TDE Work?

Other encryption methodologies encrypt data prior to it being inserted or updated in the table. The encryption could happen in the client application, the middle-tier code, or in the database server itself using the new encryption abilities that were introduced in SQL Server 2005. The key point here is that the data is encrypted before it even gets put into cache (and hence before it is written to disk) and stays encrypted in cache and on disk. Decryption conversely occurs when the data is retrieved from cache or in the middle tier, or in the client application.

TDE is different because it isn't encrypting some of the data prior to being written to cache (and therefore disk); it is encrypting all data, an 8K page at a time, as it is written to disk, and it is decrypting the data as it is read from disk. (See Figure 1-16.) This encryption is occurring at a low level and, although it occurs for all data pages, it only occurs as data is written to or read from disk. So in a more static database, with a low transaction count and high query count that is held mostly in cache, you will see little if no impact on performance. On a very high transactional database that is much, much bigger than could be held in cache (in other words, high disk input/output, or I/O), you are more likely to see an impact on performance.

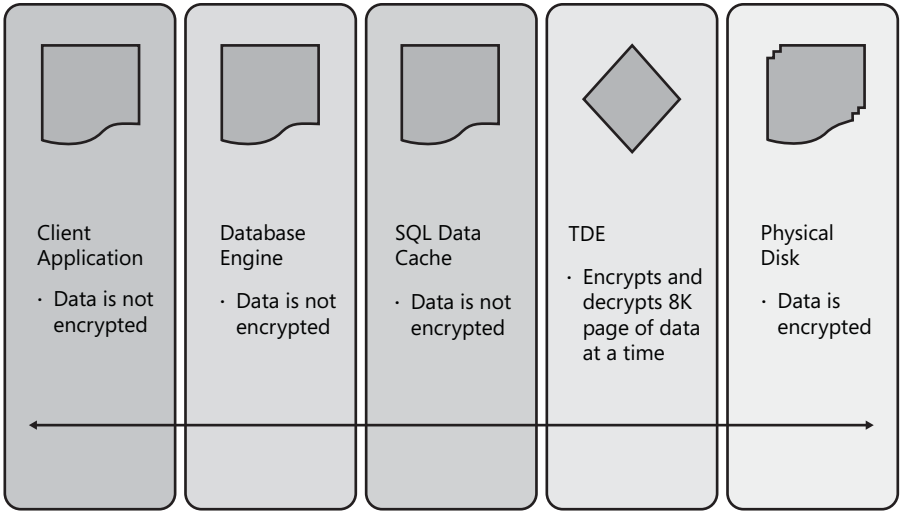


FIGURE 1-16 Transparent Data Encryption in the flow of data

Performance Considerations

You didn't think you could get such an awesome feature without some caveats? Alas, there are some things that you must take into consideration before encrypting your database. Performance and security are always at odds with one another, and TDE is no exception to this rule. If you want to secure your data, you need to do extra processing to encrypt it. You may decide only to encrypt certain individual columns of data as opposed to using TDE. As I stated earlier in the chapter, the methodology you use depends on your specific situation. Knowing all the facts ahead of time (such as exactly what data needs to be secure, how active the database is, and how big the database is) will make planning easier and allow you to make informed decisions about using TDE. I do not want to give the impression that TDE is slow. On the contrary, TDE will usually be faster than other encryption methodologies available within SQL Server, even when only a very little data would normally be encrypted. This speed is a result of a number of factors, including the fact that TDE encrypts data only as it travels back and forth from disk to cache (so data is not constantly being encrypted and de-

crypted but rather only as it needs to be) and that data in cache is unencrypted, which means fully efficient query optimizations and use of indexes.

Tempdb and Transparent Data Encryption

If any database on your SQL Server instance uses TDE, then tempdb also automatically gets encrypted on that instance. That means that if on a SQL Server instance you have 10 databases, 9 of which are not encrypted and 1 of which is encrypted, then you will be using an encrypted tempdb for all 10 databases. This could have a great impact on those other 9 databases if they happen to make much use of tempdb.

If any of those 9 unencrypted databases do use tempdb a lot and the 1 encrypted database does not use tempdb much, then you should consider using a technology other than TDE to encrypt that 1 database or consider separating the encrypted and non-encrypted databases in different SQL Server instances.

Certificate and Key Management

I also need to inform you about the other caveat: certificate and key management. Sure, you now have a great way to keep your data secure in case someone steals a backup or acquires a copy of a detached database, but what about you, when you realize you need to restore the database to another server? Do you have the appropriate keys or certificates to restore and use the database?

Let's examine an example that will encrypt the database *AdventureWorks2* (a copy of *AdventureWorks* with which I can, let's say, "experiment"). This example will first create the master key and a certificate in the master database; it will continue to create the database encryption key (a special key in a database used for TDE) and then turn on TDE for the *AdventureWorks2* database.

```
USE master;
GO

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'P@sswOrd!!';
GO

CREATE CERTIFICATE MyServerCert WITH SUBJECT = 'My Server Certificate'
GO

USE AdventureWorks2
GO

CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
```

```
ENCRYPTION BY SERVER CERTIFICATE MyServerCert
GO

ALTER DATABASE AdventureWorks2
SET ENCRYPTION ON
GO
```

You should immediately run the following query:

```
;WITH es (encryption_state, encryption_state_desc)
AS(    SELECT *
    FROM
    (VALUES
        (0, 'No database encryption key present, no encryption'),
        (1, 'Unencrypted'),
        (2, 'Encryption in progress'),
        (3, 'Encrypted'),
        (4, 'Key change in progress'),
        (5, 'Decryption in progress'))
    EncryptState (encryption_state, encryption_state_desc)
    )
SELECT db_name(dek.database_id) as database_name
    ,es.encryption_state_desc
    ,dek.percent_complete
    ,dek.key_algorithm
    ,dek.key_length
FROM sys.dm_database_encryption_keys AS dek
 INNER JOIN es ON es.encryption_state = dek.encryption_state
GO
```

If you waited too long to run the query, the encryption may complete, and the result would look like this, instead:

database_name	encryption_state_desc
Tempdb	Encrypted
AdventureWorks2	Encrypted

But if you ran the query quickly enough, you will see results similar to the following (the percent_complete value for *AdventureWorks2* will vary).

database_name	encryption_state_desc	percent_complete	key_algorithm	key_length
Tempdb	Encrypted	0	AES	256
AdventureWorks2	Encryption in progress	14.04779	AES	128

You will also notice two interesting things about these results:

- Tempdb is also in an encrypted state. Prior to encrypting *AdventureWorks2*, no user databases were encrypted on this server instance, so tempdb was not encrypted. This query would have returned no rows. However, as mentioned earlier in this chapter, once you turn on TDE for any database in a server instance, tempdb also gets encrypted.
- Tempdb is using the same key algorithm but a longer key length (the maximum key length for AES, actually). In reality, even if you had used Triple Data Encryption Standard (DES) for your database encryption key algorithm, tempdb would still (and always) be encrypted using AES 256 because it is the most secure of the encryption algorithms.

Now, what would happen if we made a backup of *AdventureWorks2*, dropped the database, re-created the database, and then finally restored it from the backup? Well, because we are on the same server instance, it would work fine, but if we were on a different server instance, or if we "accidentally" dropped the server certificate, MyServerCert, that was used to create the database encryption key, we would have problems. Let's go through the latter scenario on this server instance.

Examine this script that makes the backup or the database and the server certificate (we'll need that later to fix the problem we will introduce), then drops the certificate and attempts to restore the database:

```
BACKUP DATABASE [AdventureWorks2]
TO DISK =
  N'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Backup\AdventureWorks2.bak'
WITH
  NOFORMAT, INIT, NAME = N'AdventureWorks2-Full Database Backup',
  SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO

USE MASTER
GO

BACKUP CERTIFICATE MyServerCert TO FILE = 'c:\MyServerCert.cer'
WITH PRIVATE KEY
  (FILE = 'c:\MyServerCert.key', ENCRYPTION BY PASSWORD = 'p455w0rd');
GO

DROP CERTIFICATE MyServerCert
GO

RESTORE DATABASE [AdventureWorks2]
FROM DISK =
  N'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Backup\AdventureWorks2.bak'
WITH
  FILE = 1, NOUNLOAD, REPLACE, STATS = 10
GO
```

This will fail on the last step, restoring the database, because without the server certificate that was used to create the database encryption key, the key is not usable. Therefore, no restore is possible. If this were a different server instance (some thief's server, perhaps), the restore would not be possible, and your data would be safe.

However, what if it was your data that you needed to recover? You would need the server certificate, which you can re-create from a backup (made in the previous script), and then you could successfully restore the database, as shown here:

```
CREATE CERTIFICATE MyServerCert
FROM FILE = 'c:\MyServerCert.cer'
WITH PRIVATE KEY
  (FILE = 'c:\MyServerCert.key', DECRYPTION BY PASSWORD = 'p455w0rd');
GO

RESTORE DATABASE [AdventureWorks2]
FROM DISK =
  N'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Backup\AdventureWorks2.bak'
WITH
  FILE = 1, NOUNLOAD, REPLACE, STATS = 10
GO
```

Extensible Key Management

In SQL Server 2005, new certificate and key functionality was added that allowed you to define and use certificates, asymmetric keys, and symmetric keys in the database. This functionality has been extended in SQL Server 2008 with the advent of Extensible Key Management (EKM).

EKM allows a third-party vendor to register a library that allows access to its Hardware Security Module (HSM), a software- or hardware-based device for storing keys, part of an enterprise key management solution.

For example, a third-party vendor may have a centralized smart card system that can allow access to a single smart card device from any server in the domain. The vendor provides a library that can be installed on a server, which gives it the ability to access that centralized smart card device. SQL Server 2008 can register this library as a provider and subsequently access it. So now all your SQL Server 2008 instances can use a centralized set of keys. Encrypted data shared across instances can be more easily used, because you don't need to go through the more cumbersome process of duplicating keys amongst instances. Instead, each can use a key encrypted by a key from the EKM provider.

Key management is a complex topic that is well beyond the scope of this book. However, we can walk through a simple example that shows how an EKM could be used to encrypt data in your database.

EKM in Practice

To use the EKM, you must first register it in SQL Server as a cryptographic provider. For example, if we had an EKM provider library named KeyMgrPro.dll, it could be registered in SQL as follows:

```
CREATE CRYPTOGRAPHIC PROVIDER cpKeyManagerPro
    FROM FILE = 'C:\Program Files\Key Manager Pro\KeyMgrPro.dll'
```

Now that we have registered the EKM in SQL Server, the next step is to use it to secure your data. The first step is to create a symmetric or asymmetric key using a key on the EKM cryptographic provider:

```
CREATE SYMMETRIC KEY CreditCardSymKey
FROM PROVIDER cpKeyManagerPro
WITH
    PROVIDER_KEY_NAME='CreditCardKey',
    CREATION_DISPOSITION=OPEN_EXISTING;
```

This example uses an existing key named CreditCardKey on the cpKeyManagerPro cryptographic provider. At this point, you have a symmetric key, and you can use it just like you have used a symmetric key since SQL Server 2005.

Now suppose you add a new column to the Sales.CreditCard table that will hold the encrypted credit card number, as shown here:

```
ALTER TABLE Sales.CreditCard
    ADD EncryptedCreditCardNumber varbinary(128);
```

You can then populate the encrypted values from the existing credit card numbers.

```
UPDATE Sales.CreditCard
SET EncryptedCreditCardNumber =
    EncryptByKey(
        Key_GUID('CreditCardSymKey'),
        CreditCardNumber,
        1,
        HashBytes('SHA1', CAST(CreditCardID AS VARBINARY)));
```

At this point, you no longer need the unencrypted credit card number, and you can use the symmetric key to decrypt a credit card as needed.

```
SELECT CreditCardID,
    CAST(DecryptByKey(
            EncryptedCreditCardNumber,
            1, HashBytes('SHA1', CONVERT(CreditCardID AS VARBINARY)))
        AS NVARCHAR) AS CreditCardNumber
FROM Sales.CreditCard = @CreditCardID;
```

You would incorporate this into a function and use it to decrypt the credit card number for a specific card, or perhaps you would use this in a stored procedure. The beauty is that you don't need to open and close the symmetric key, as that is handled by the EKM cryptographic provider.

Summary

On the manageability front, Policy-Based Management lessens the work previously required to keep servers, databases, and an assortment of objects in conformity with company policies. There are other features that have not yet been discussed, such as the advanced edit capability for conditions, which allows you to create custom expressions using the available facet properties and a variety of built-in functions (including the ability to dynamically execute T-SQL). I hope that this gives you enough to become familiar with the workings of the policy management and to continue increasing your knowledge.

SQL Server 2008 Books Online has additional examples and step-by-step instructions on how to create several different policies. I recommend reading through these and trying them out as they can give types of examples other than those covered in this chapter.

On the security front, TDE is a phenomenal technology that allows you to quite literally transparently encrypt your data as it is written to disk and decrypt it as it is read from disk. Because all data, in data files and transaction log files, are encrypted, you can feel safer about what happens to your data after it leaves for offsite storage, while it's in transport to a client or consultant, or if someone unscrupulous gets his or her hands on the data files.

It also works in a very efficient manner, only performing encryption as data goes from cache to disk and vice versa. Although it seems like overkill, it really only does the work as needed, allowing for an efficient way to secure your data.

And finally, on the security and manageability front, you have auditing and EKM: Auditing to let you see what has been happening on your servers and for EKM to allow you to use keys from your enterprise key management solution instead of individually managing keys on each SQL Server instance.

You now have a more secure and easier to manage database server.

Chapter 2

Performance

SQL Server 2008 has some new and some improved features to better performance and to more easily monitor performance. In this chapter, we'll be discussing the new Resource Governor, which allows you to govern the amount of resources an application can use. You will see how data and backup compression can help reduce the storage requirements of your data and thus reduce the amount of disk input/output (I/O) to help better performance. You will also explore the new data collection capabilities of SQL Server 2008, which allows you to collect trace, query, and Performance Monitor data to help you evaluate the performance aspects of your database server. And you will look at new query plan freezing features that give you more flexibility and options for working with plan guides to help better performance.

Resource Governor

We've all seen it before. One poorly written application slows your SQL Server to a crawl, consuming far more resources than it should and, in the process, hurting the performance of other more mission-critical applications. And what could you do except admonish those who wrote the bad application in hopes it would be corrected and apologize to those using the mission-critical applications?

Well no more! SQL Server 2008 can now give those mission-critical applications the resources they deserve and that less critical application fewer resources so that it doesn't hurt the overall performance of the database server instance. Resource Governor, a new addition to SQL Server 2008, can allow you to group different resources and allocate more or less resources depending on the requirements for the applications.

Resource Pools

A resource pool represents a possible subset of the server's resources. By default, there are two resource pools. The first is used internally for resources required by SQL Server itself and is aptly named "internal." The second, named "default," is the resource pool used by incoming sessions. Because, initially, there are no other resource pools, all incoming requests would be directed to the default resource pool. This behavior of all incoming requests sharing a single resource pool is the same behavior you currently see in SQL Server 2005 and earlier, where all incoming requests are grouped together and treated as equals.

Resource Pool Settings

Besides the name, a resource pool has four settings that can be specified:

Setting	Name	Description
Minimum CPU %	MIN_CPU_PERCENT	The guaranteed average amount of CPU time for all requests being serviced by the resource pool when there is contention for CPU time. It can be a value between 0 and 100, and, for all resource pools, the sum of this value must be less than or equal to 100.
Maximum CPU %	MAX_CPU_PERCENT	The maximum average amount of CPU time for all requests being serviced by the resource pool when there is contention for CPU time. This value must be greater than the Minimum CPU % and less than or equal to 100.
Minimum Memory %	MIN_MEMORY_PERCENT	The minimum amount of memory that is dedicated to this resource pool (not shared with other resource pools). Valid values are from 0 to 100. For all resource pools, the sum of this value must be less than or equal to 100.
Maximum Memory %	MAX_MEMORY_PERCENT	The maximum amount of memory that can be used by this resource pool. This value must be greater than the Minimum Memory % and less than or equal to 100.

Both minimum settings have a default value of 0, and both maximum settings have a default value of 100. All four settings are optionally specified when creating a resource pool. Also note that, for all pools except the internal pool, if the sum of all the minimum percent (of either CPU or memory) is 100, then each pool will use at most its minimum percent value. For example, if you have 3 resource pools—default, MonitorAppPool, and RealtimeEventsPool—each with a respective Minimum CPU % of 20, 30, and 50, then the default pool will never use more than 20 percent (when there is contention for CPU), because MonitorAppPool and RealtimeEventsPool require 80 percent of the CPU resources.

This realized maximum percentage is known as the Effective Maximum Percent and represents the maximum amount of resources the pool can have when resources are in contention. For any given pool (except the internal pool, which is not counted in this calculation), this effective maximum can be calculated using the following formula (for CPU or memory):

Effective Maximum % = 100 – MIN(Max % value, SUM(All Other Min % values))

Another calculated value of note is the Shared Percentage, which is simply the difference between the Effective Maximum % and the Min %. This value determines how much of the

pool's resources are sharable when resources are in contention. For example, examine the following table (copied directly from an Excel workbook that I used to calculate these values).

Pool Name	Min CPU %	Max CPU %	Effective Max CPU %	Shared CPU %
Internal	0	100	100	0
Default	20	100	50	30
MonitorAppPool	25	50	50	25
RealtimeEventsPool	25	75	55	30

Although you would think that MonitorAppPool and RealtimeEventsPool should have the same Effective Maximum CPU %, MonitorAppPool is actually lower (50 versus the 55 value of RealtimeEventsPool) because MonitorAppPool also has a Maximum CPU % of 50, which limits its Effective Maximum CPU %.

To sum up things: When resources are in contention, a pool will use between its Minimum Percent and its Effective Maximum Percent.

Note Resource pool settings are applicable to each CPU available for use by SQL Server. There is no configuration to associate a resource pool with a particular CPU.

Workload Groups

Incoming requests are not directly associated to a resource pool. Instead, workload groups are used to allow for a subdivision of a pool's resources, as well as a means of associating requests to resources. A resource pool can be associated with multiple workload groups, but a workload group may be associated with one and only one resource pool.

Workload groups allow you to further separate tasks within a resource pool and are the basis for the use of the classifier function (discussed later in this section), which determines to which workload group an incoming request is associated.

When you associate multiple workload groups with the same resource pool, you can further define how each behaves with the resource pool. For example, you can tell a particular workload group to get preferential treatment within the resource pool by assigning the IMPORTANCE setting to a value of LOW, MEDIUM (the default), or HIGH. You can also limit the amount of memory used relative to the other groups in the resource pool by changing the REQUEST_MAX_MEMORY_GRANT_PERCENT setting (the default is 25).

Important Setting REQUEST_MAX_MEMORY_GRANT_PERCENT to a value greater than 50 will result in large queries running one at a time.

For information on the other workload group settings, see the topic "CREATE WORKLOAD GROUP (Transact-SQL)" in SQL Server Books Online.

The Classifier Function

By default, the Resource Governor doesn't specify a classifier function, so if it is enabled in this state, all requests are directed to the default workload group, which is part of the default resource pool. And so, user-defined resource pools or workload groups will not have any incoming requests associated with them until you implement and assign a classifier function that can associate the requests with the groups.

A classifier function is simply a scalar schema-bound, user-defined function with no parameters in the master database that is used to determine with which workload group the session should be associated. It is evaluated for each new incoming session (regardless of connection pooling settings), and its return value is literally the name of the workload group that should be used for the session. If the function returns null or the name of a non-existent workload group, the default workload group is used for the session.

Note If you create a resource pool that has a minimum but is never associated with an incoming request, you can effectively waste memory or CPU because that minimum will reduce the effective maximum of the other resource pools.

The following example simply checks the login name, and, if it is either MonitorServiceUser or RealtimeServiceUser, then it returns the appropriate workload group name. If the login name doesn't match either of these values, the function will return null (the CASE expression returns null).

```
USE [master]
GO
CREATE FUNCTION [dbo].[fnClassifier]()
RETURNS SYSNAME WITH SCHEMABINDING
BEGIN
    RETURN
        CAST(CASE SUSER_SNAME()
                WHEN 'MonitorServiceUser' THEN 'MonitorGroup'
                WHEN 'RealtimeServiceUser' THEN 'RealtimeGroup'
            END AS SYSNAME)
END
GO
```

The next example uses the application name to determine the associated workload group.

```
CREATE FUNCTION [dbo].[fnClassifyByApp]()
RETURNS SYSNAME WITH SCHEMABINDING
BEGIN
    RETURN
        CAST(
            CASE APP_NAME()
                WHEN 'MonitorApp' THEN 'MonitorGroup'
                WHEN 'DashboardApp' THEN 'MonitorGroup'
                WHEN 'RealtimeApp' THEN 'RealtimeGroup'
            END AS SYSNAME)
END
GO
```

Although the login name may not be easily changed, other values such as workstation id (HOST_NAME) or the application name (APP_NAME) can be changed in the connection string used to connect to the server. So if your application needs to change its associated workload group, you can simply change the application name value in the connection string.

```
Data Source=YourServer;Initial Catalog=YourDatabase;Persist Security
Info=True;Trusted_Connection=True;Application Name=MonitorApp;
```

Note Although using the workstation id or the application name is a simple way to change a request's associated workload group without requiring any application code changes, it also can invite potential security issues. For example, a malicious user could use this method to associate an application to a workload group with more resources (i.e., higher maximums and minimums), giving the application more resources than originally intended or allowed to have. Therefore, this method of associating requests to workload groups should only be used if you trust all the applications that are connecting to your database server instance.

You could also have a table containing a list of application name values (or login names or other information to group incoming requests) and an associated workload group, and you could use it as the source for your classifier function, for example.

```
CREATE TABLE dbo.AppWorkloadGroups
(
    AppName NVARCHAR(128) NOT NULL PRIMARY KEY,
    WorkloadGroup SYSNAME NOT NULL
)
GO
```

```
INSERT INTO dbo.AppWorkloadGroups
VALUES
    (N'MonitorApp', 'MonitorGroup')
    , (N'DashboardApp', 'MonitorGroup')
    , (N'RealtimeApp', 'RealtimeGroup')
GO

CREATE FUNCTION [dbo].[fnClassifyWorkloadGroupByApp]()
RETURNS SYSNAME WITH SCHEMABINDING

BEGIN
    RETURN
            CAST(
                    (SELECT MIN(WorkloadGroup)
                    FROM dbo.AppWorkloadGroups
                    WHERE AppName = (APP_NAME()))
            AS SYSNAME)
END
GO

ALTER RESOURCE GOVERNOR
WITH (CLASSIFIER_FUNCTION = [dbo].[fnClassifyWorkloadGroupByApp]);
GO

ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Now you can more easily manage how applications get associated to workload groups.

Note Dedicated administrator connections (DACs) are not affected by the Resource Governor, regardless of the classification function, because DACs always run in the internal workload group. So the DAC can be used to troubleshoot issues with the classifier function.

Creating Resource Pools and Workload Groups

You can create resource pools and workload groups using either Transact-SQL (T-SQL) or SQL Server Management Studio. This first example shows how to create two resource pools. The first resource pool is for monitoring related applications and includes two workload groups: one for the live monitor application used by IT staff (high importance) and one for a dashboard-style application used by corporate executives (lower importance), which should give way to the monitor application as needed.

The second resource pool is for the real-time event collection services. It should be given more resources than the monitoring application resource pool (to ensure proper data collection). It contains a single workload group. This following example is a continuation of the classifier function example from the previous code listing.

```
CREATE RESOURCE POOL [MonitorAppPool]
WITH (MAX_CPU_PERCENT=25)
GO

CREATE WORKLOAD GROUP [MonitorGroup]
WITH (IMPORTANCE=HIGH)
USING [MonitorAppPool]

GO

CREATE WORKLOAD GROUP [DashboardGroup]
WITH (IMPORTANCE=LOW
    , REQUEST_MAX_MEMORY_GRANT_PERCENT=10)
USING [MonitorAppPool]
GO

CREATE RESOURCE POOL [RealtimeAppPool]
WITH (MIN_CPU_PERCENT=25
    , MAX_CPU_PERCENT=75)
GO

CREATE WORKLOAD GROUP [RealtimeGroup]
WITH (IMPORTANCE=HIGH)
USING [RealtimeAppPool]
GO
```

Using SQL Server Management Studio (SSMS), you can manage the state and associated classifier function of the Resource Governor, as well as all resource pools, workload groups, and their respective settings all in a single dialog box. Figure 2-1 shows the example above done with SSMS.

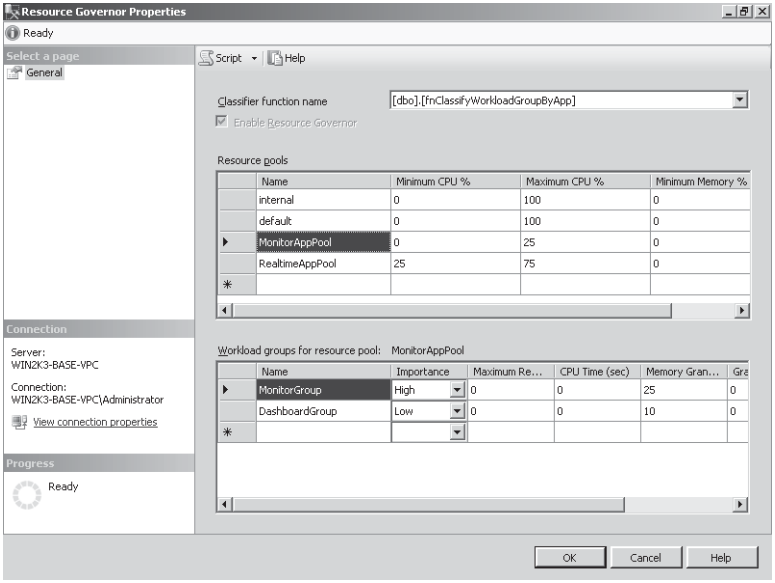


FIGURE 2-1 Resource Governor Properties dialog box

Data and Backup Compression

SQL Server Enterprise (and thus Developer) edition now allows you to effortlessly implement data and backup compression.

Data Compression

Data compression can compress table data, nonclustered indexes, and indexed views. Of course, because a clustered index contains the table data in its leaf level, clustered indexes are compressed when a table is compressed. There are no changes required to your code to access data that is in a compressed state because the compression and decompression is done behind the scenes.

Row Compression

Row compression uses variable length storage to reduce row storage requirements. Not all data types can be compressed. For example, an INT of 4 bytes can be reduced such that only the bytes needed for storage are used. So if an INT column has a value of 12,345, then it will

use only 2 bytes (because a 2 byte integer can hold values from –32,768 to 32,767). The topic "Row Compression Implementation" in SQL Server Books Online describes all the data types and if and how much they can be compressed.

Note When compression is implemented, 0 (numeric) and NULL values do not take up any space.

Page Compression

Page compression is actually a combination of three different compression types: row, prefix, and dictionary. First a row compression operation is done, in an attempt to reduce the data type storage.

Note When using page compression on a nonclustered index, the non-leaf nodes of the index are compressed using only row compression. The leaf nodes use all of the three compressions (discussed in this section).

Next, a prefix compression operation is performed. Prefix compression looks for patterns in the start of the data and repeated prefix values in the page header. For example, review the pieces of data shown in Figure 2-2.

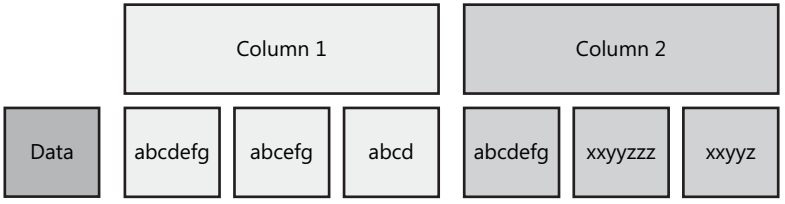


FIGURE 2-2 Data before compression

Note The prefix and dictionary values are actually stored in a compression information (CI) structure in an area just after the page header.

When prefix compression is applied, different initial patterns within the same column are moved to the CI structure, and the data is changed to reference the values in the CI structure (pointers are represented here by the equal border color), as shown in Figure 2-3.

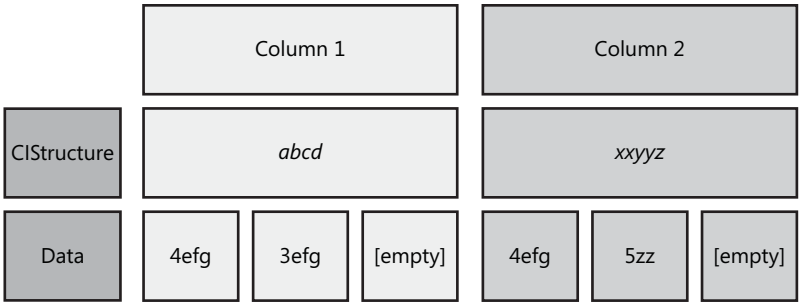


FIGURE 2-3 Data after prefix compression

A value of [empty] in the data means that data is an exact match and just a pointer to the prefix value in the CI structure is stored. The value 4efg replaces the prefix of abcd (four characters) and points to the prefix value in the CI structure. The value 3efg only uses three of the characters from the prefix in the CI structure.

Now that the prefix compression has been completed, a third compression operation occurs. Dictionary compression looks for repeated patterns throughout the data and replaces them, as shown in Figure 2-4.

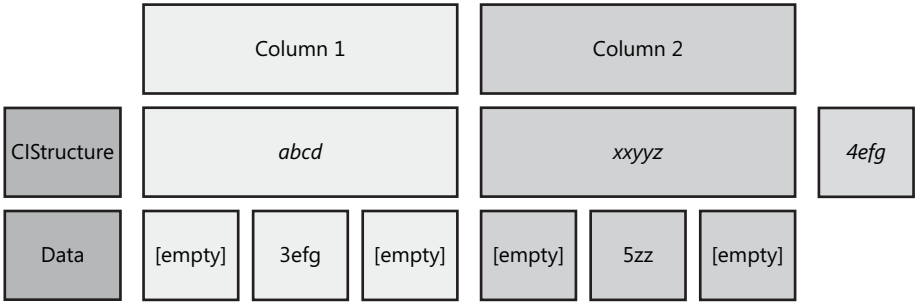


FIGURE 2-4 Data after dictionary compression

For example, the [empty] value in the box on the left of Column 2 represents the value 4efg, which is stored in the CI structure. In other words, dictionary compression consolidates data across columns.

Now that you have seen the three compression operations that occur for page compression, I must tell you that, on new pages, the process is a little different. Each new page only uses row compression until the page is full and another row is then attempted to be added. At this point, the engine must evaluate the columns for prefix compression and the page for dictionary compression. If the compression gains enough space (less the CI overhead) for

another row to be added, the data is page compressed in addition to the existing row compression. If not enough space is gained, no page compression occurs, and only the row compression remains in effect.

Implementing Data Compression

So how do you actually compress the data in a table or a nonclustered index? As usual, you have two choices: use SSMS or use T-SQL.

SSMS provides a wizard interface to generate or execute a T-SQL script. The first step is to start the wizard. As shown in Figure 2-5, you navigate to a table in the Object Explorer, right-click on the table, choose Storage, and then choose Manage Compression to get things started.

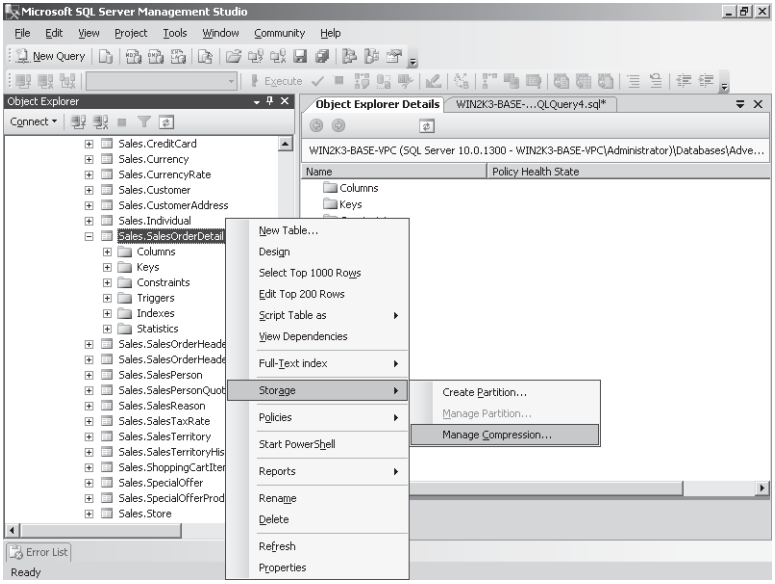


FIGURE 2-5 Manage Compression context menu

You will then see a Welcome screen (that you optionally choose not to display on subsequent visits to the wizard). The Data Compression Wizard, as shown in Figure 2-6, allows you to set up the compression individually on each partition. In this example, there is only one partition, but a table may be spread across several partitions. You can choose to use the same setting for all of them (by selecting the Use Same Compression Type For All Partitions check box) or to individually choose page compression, row compression, or no compression. This step in the wizard also lets you calculate the compressed space size (by clicking the Calculate button after selecting your compression options).

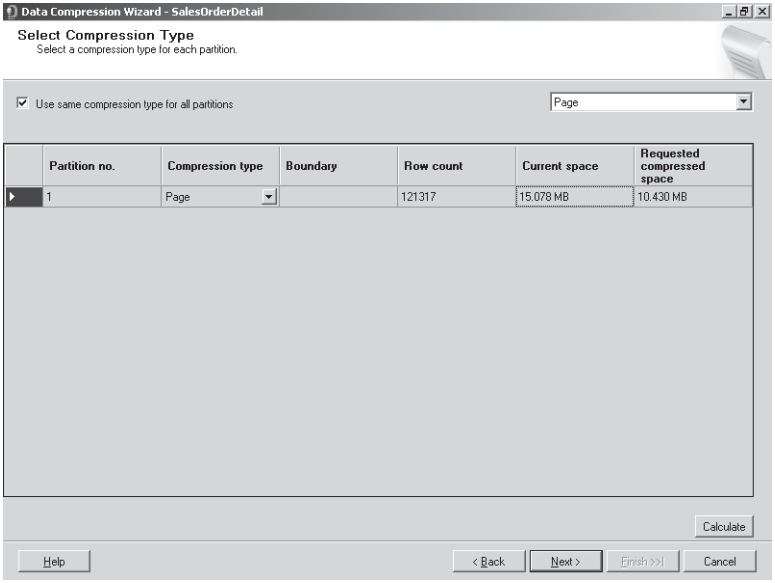


FIGURE 2-6 Selecting the compression type in the Data Compression Wizard

At this point, you are ready to either create a script, execute the script, or schedule the script to be executed. In Figure 2-7, I am opting to generate the script in a new query window.

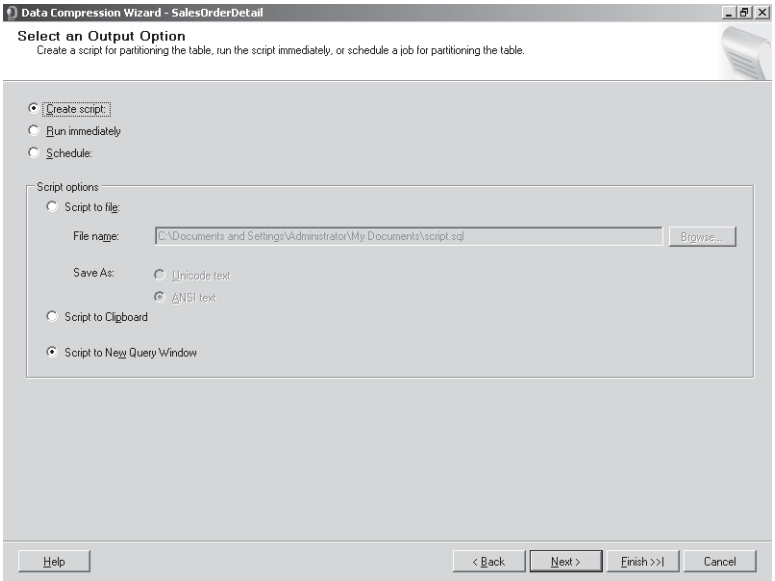


FIGURE 2-7 Selecting an output option in the Data Compression Wizard

The next screen is a review screen, and the last is a status of the script generation (and optional execution). When all is said and done, I end up with the following code in a new query window.

```
USE [AdventureWorks]
ALTER TABLE [Sales].[SalesOrderDetail] REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = PAGE)
```

You can, of course, choose to write the T-SQL instead of using the wizard, but the additional scheduling options done via the wizard are a nice plus. Also, if you are still unfamiliar with the new syntax, or if you have a more complex structure, the wizard can make the work much easier.

Compressing Partitions

Compressing data in partitions is very similar to compressing data in a single partition table. For reference, this code adds four filegroups, four files, a partition function, and a partition scheme, and it changes the SalesOrderDetail table to be partitioned (all of the code was generated via SSMS wizards or property windows).

```
USE [master]
GO
ALTER DATABASE [AdventureWorks] ADD FILEGROUP [FG1]
GO
ALTER DATABASE [AdventureWorks] ADD FILEGROUP [FG2]
GO
ALTER DATABASE [AdventureWorks] ADD FILEGROUP [FG3]
GO
ALTER DATABASE [AdventureWorks] ADD FILEGROUP [FG4]
GO

ALTER DATABASE [AdventureWorks]
ADD FILE ( NAME = N'File1', FILENAME = N'C:\Program Files\Microsoft SQL Server\
MSSQL10.MSSQLSERVER\MSSQL\DATA\File1.ndf' , SIZE = 1024OKB , FILEGROWTH = 1024KB )
TO FILEGROUP [FG1]
GO
ALTER DATABASE [AdventureWorks]
ADD FILE ( NAME = N'File2', FILENAME = N'C:\Program Files\Microsoft SQL Server\
MSSQL10.MSSQLSERVER\MSSQL\DATA\File2.ndf' , SIZE = 1024OKB , FILEGROWTH = 1024KB )
TO FILEGROUP [FG2]
GO
ALTER DATABASE [AdventureWorks]
ADD FILE ( NAME = N'File3', FILENAME = N'C:\Program Files\Microsoft SQL Server\
MSSQL10.MSSQLSERVER\MSSQL\DATA\File3.ndf' , SIZE = 1024OKB , FILEGROWTH = 1024KB )
```

```
TO FILEGROUP [FG3]
GO
ALTER DATABASE [AdventureWorks]

ADD FILE ( NAME = N'File4', FILENAME = N'C:\Program Files\Microsoft SQL Server\
MSSQL10.MSSQLSERVER\MSSQL\DATA\File4.ndf' , SIZE = 10240KB , FILEGROWTH = 1024KB )
TO FILEGROUP [FG4]
GO

USE [AdventureWorks]
GO
BEGIN TRANSACTION
CREATE PARTITION FUNCTION [fnSalesOrderDetail_PartitionBySalesOrderID](int)
AS RANGE LEFT FOR VALUES (N'50000', N'65000', N'80000')

CREATE PARTITION SCHEME [psSalesOrderDetail]
AS PARTITION [fnSalesOrderDetail_PartitionBySalesOrderID]
TO ([FG1], [FG2], [FG3], [FG4])

ALTER TABLE [Sales].[SalesOrderDetail]
DROP CONSTRAINT [PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID]

ALTER TABLE [Sales].[SalesOrderDetail]
ADD  CONSTRAINT [PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID] PRIMARY KEY
CLUSTERED
(
    [SalesOrderID] ASC,
    [SalesOrderDetailID] ASC
)ON [psSalesOrderDetail]([SalesOrderID])

COMMIT TRANSACTION
```

Now, let's say you want to set up data compression on the SalesOrderDetail table with the following settings:

- Partition 1 with no compression
- Partition 2 with row compression
- Partitions 3 and 4 with page compression

You could run the wizard and set the values shown in Figure 2-8.

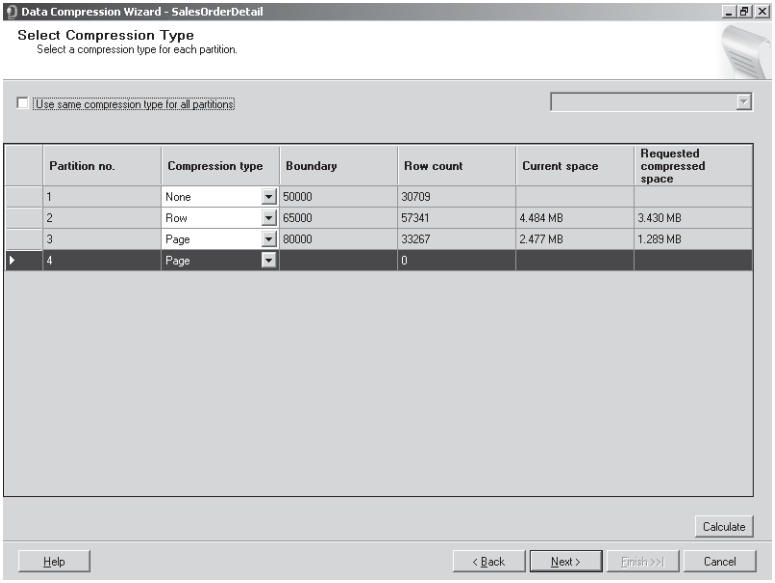


FIGURE 2-8 Selecting compression type for partitioned table in the Data Compression Wizard

Or you can write the following T-SQL code (which was, of course, generated from the wizard).

```
USE [AdventureWorks]
GO
ALTER TABLE [Sales].[SalesOrderDetail] REBUILD PARTITION = 2
WITH(DATA_COMPRESSION = ROW )
USE [AdventureWorks]
ALTER TABLE [Sales].[SalesOrderDetail] REBUILD PARTITION = 3
WITH(DATA_COMPRESSION = PAGE )
USE [AdventureWorks]
ALTER TABLE [Sales].[SalesOrderDetail] REBUILD PARTITION = 4
WITH(DATA_COMPRESSION = PAGE )
GO
```

As promised, this process is very similar to a table on a single partition.

Compressing Nonclustered Indexes

As you likely already determined, compressing a nonclustered index is also a very similar process. You can, of course, right-click on the index in Object Explorer, choose Storage, and then

choose Manage Compression, and use the Data Compression Wizard, or you can simply write T-SQL code, as shown here.

```
USE [AdventureWorks]
ALTER INDEX [IX_SalesOrderDetail_ProductID] ON [Sales].[SalesOrderDetail]
REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = PAGE )
```

Backup Compression

Compressing backup data is not so much about saving disk space as it is about reducing disk I/O. Although backup compression can potentially save a lot of disk I/O, this savings does come at a price—an increase in CPU usage. Overall, however, backup compression will usually result in better performance. I will revisit the CPU performance increase later in this section.

Configuring and Using Backup Compression

Backup compression is off by default. You have two options that allow you to use or not use backup compression: change the default server configuration value for backup compression or override this default configuration for individual backups.

You can change the default configuration by using sp_configure, as shown here.

```
EXEC sp_configure 'backup compression default', 1
RECONFIGURE WITH OVERRIDE;
```

Setting this configuration setting to a value of 1 means all backups will be compressed unless specifically told not to be compressed in the BACKUP statement, and a value of 0 means backups will not be compressed, unless specifically told to be compressed. For example, if the default setting was 0, but you wanted the AdventureWorks database backup to be compressed, you simply add the COMPRESSION option, as shown here.

```
BACKUP DATABASE AdventureWorks
TO DISK='C:\AdvWorksData.bak' -- C Drive for demo purposes only
WITH FORMAT,
     COMPRESSION,
     NAME='AdventureWorks Full Compressed Backup'
GO
```

If you do not want a database backup to be compressed, you explicitly state it using the NO_COMPRESSION option.

You can check the compression ratio of your backup by checking the ratio of backup_size and compressed_backup_size in the backupset history table (in msdb database).

```
SELECT name,
    backup_size,
    compressed_backup_size,
    backup_size/compressed_backup_size as ratio
FROM msdb..backupset
WHERE name = 'AdventureWorks Full Compressed Backup'
GO
```

This query gives the following results (individual compression size may vary).

name	backup_size	compressed_backup_size	Ratio
AdventureWorks Full Compressed Backup	183583744	42139433	4.356578409586099556

The ratio is about 4.36 to 1, meaning that the compressed backup is using just under 23 percent of the space that the equivalent uncompressed backup would have used.

Using Resource Governor to Minimize CPU Impact

As I mentioned earlier in this section, backup compression requires more CPU processing (in exchange for less disk I/O). This increase in CPU usage can be mitigated by using Resource Governor to limit the maximum amount of CPU used by backup compression.

```
-- Create login
USE master;
GO
CREATE LOGIN [Domain\BackupUser] FROM WINDOWS
GRANT VIEW SERVER STATE TO [Domain\BackupUser]
GO

-- Add user to database
USE AdventureWorks;
GO
CREATE USER [Domain\BackupUser] FOR LOGIN [Domain\BackupUser]
EXEC sp_addrolemember 'db_backupoperator', 'Domain\BackupUser'
GO
```

```
-- Create resource pool and workload group
USE master;
GO
CREATE RESOURCE POOL [BackupPool]
WITH (MAX_CPU_PERCENT=20)
GO
CREATE WORKLOAD GROUP [BackupGroup]
USING [BackupPool]
GO

-- Create classifier function
CREATE FUNCTION [dbo].[fnClassifier]()
RETURNS SYSNAME
WITH SCHEMABINDING
BEGIN
    RETURN
        CAST(
            CASE SUSER_SNAME()
                WHEN 'Domain\BackupUser' THEN 'BackupGroup'
            END
        AS SYSNAME)
END
GO

-- Associate classifier with Resource Governor
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION=dbo.fnClassifier);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

If you already have a classifier function, you probably can't simply replace it with the example above, and instead you will likely want to alter it to perform the check for the login that will perform the backups, as shown in the following code listing.

```
-- Original classifier function
CREATE FUNCTION [dbo].[fnClassifyByApp]()
RETURNS SYSNAME WITH SCHEMABINDING
BEGIN
    RETURN
        CAST(
            CASE APP_NAME()
                WHEN 'MonitorApp' THEN 'MonitorGroup'
                WHEN 'DashboardApp' THEN 'MonitorGroup'
                WHEN 'RealtimeApp' THEN 'RealtimeGroup'
            END AS SYSNAME)
END
GO
```

```
-- Altered classifier function to accommodate backup process login
ALTER FUNCTION [dbo].[fnClassifyByApp]()
RETURNS SYSNAME WITH SCHEMABINDING
BEGIN
    RETURN
        CAST(
            CASE SUSER_SNAME()
                WHEN 'Domain\BackupUser' THEN 'BackupGroup'
                ELSE CASE APP_NAME()
                        WHEN 'MonitorApp' THEN 'MonitorGroup'
                        WHEN 'DashboardApp' THEN 'MonitorGroup'
                        WHEN 'RealtimeApp' THEN 'RealtimeGroup'

                END
            END AS SYSNAME)
END
GO
```

All that's left to do is to use the [Domain\BackupUser] account for the backup process, and it will be limited to 20 percent of the CPU when other processes require CPU as well.

Other Notes Regarding Compression

- Previous versions of SQL Server (2005 and earlier) cannot read compressed backups.
- Compression cannot be used on a database that is using Transparent Data Encryption.
- Media Sets cannot contain both compressed and uncompressed backups.
- Compression doesn't change the maximum row size of 8,060 bytes; it instead allows more rows to be stored on a page.
- Nonclustered indexes do not automatically get compressed when the table is compressed. They must be compressed independently.
- When compressed data is bulk exported, it is decompressed. When data is bulk imported into a table with data compression, the data is compressed (and requires extra CPU to do the job).
- Having compressed data does not automatically cause backups to be compressed. Backup compression is a separate operation.

There are many other tidbits and details about compression that I could list, but they are listed in the topic "Creating Compressed Tables and Indexes" in SQL Server Books Online.

Performance Data Collection

Performance Monitor, SQL Server Profiler, Dynamic Management Views (DMVs) and Functions, and T-SQL in general: What do they all have in common? The answer is that they all can be used to find potential performance problems for SQL Server. On numerous occasions, I have used these and other tools to try and pinpoint where performance problems were occurring. In the past, I have combined data from Performance Monitor log files, SQL Profiler traces, and custom queries to help determine the source of performance problems, timeouts, blocking, and so on.

Sure, other tools exist both from Microsoft and third-party vendors that use data from these and other sources that allow you to analyze this data to help uncover performance problems, but what if this ability was built into SQL Server?

That's where Performance Data Collection comes in. Instead of using a set of disparate tools to collect and analyze performance data, you can now create Data Collection Sets that will gather and store Windows Performance Monitor data, SQL Profiler traces, and results from T-SQL queries (such as queries against DMVs). This data resides in a data warehouse that can be local or remote, such that multiple servers can consolidate their performance data in a single location.

My job as a consultant just got easier.

Data Collection Setup

There are several steps you need to take to use the data collector. These steps include

1. Create logins to use with Data Collector.
2. Configure the Management Data Warehouse—this is where the collected data is stored.
3. Create proxies for SQL Server Agent, if needed.

The "Data Collection" topic in SQL Server Books Online has details on how to go about setting up the Data Collector, which I will not repeat here. Once things are set up, however, the Data Collection node in Object Explorer will become enabled and have several child nodes including Disk Usage, Query Statistics, and Server Activity, as shown in Figure 2-9.

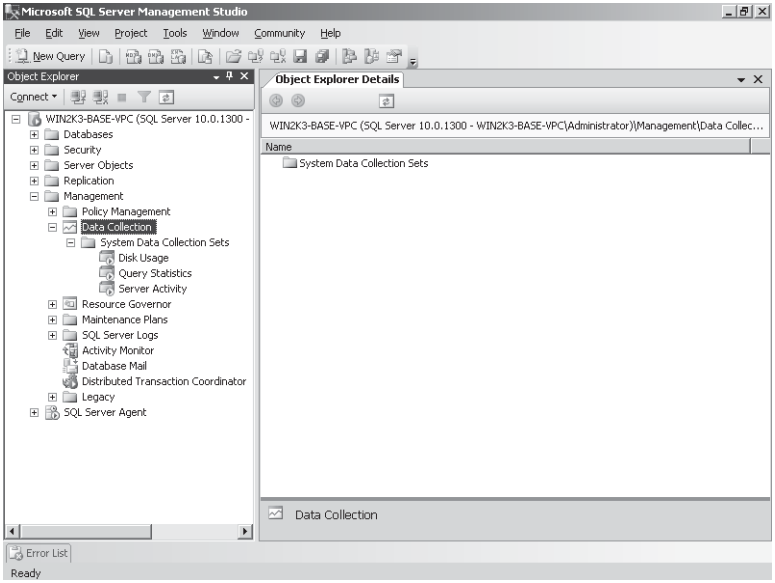


FIGURE 2-9 Data Collection in Object Explorer

Collector Types

There are four types of collectors that you can use. The three user-configurable ones will allow you to collect data from T-SQL queries, Profiler events, and Performance Monitor counters. I'll start with a brief description of each of the collector types and then follow up with a concrete example of implementing these collector types.

- **T-SQL Collector Type** This collector type allows you to use T-SQL to determine what data is collected. The code can be a simple SELECT statement or a call to a stored procedure, or it can be a more complicated block of code with variables, control-of-flow, and so on.
- **SQL Trace Collector Type** This collector type is actually doing server-side tracing, as you would do using the fn_trace_gettable() system function and the SQL Server Profiler system stored procedures (sp_trace_create, sp_trace_setevent, and so on). The trace data is collected in trace files and uploaded to the management data warehouse based on the collection set schedule, or if a manual collect and upload is requested.

  You can use the export feature SQL Profiler to take an existing trace definition and save it as a SQL Trace collection set.

- **Performance Counter Collector Type** This collector type is used to collect data from Performance Monitor counters. Unlike the sys.dm_os_performance_counters DMV, it can collect and upload not only SQL Server–related counters but also system and custom counters.
- **Query Activity Collector Type** The Query Activity Collector Type is a custom collection type used by the predefined system Query Statistics data collection set. It collects query statistics and activity from various dynamic management views, including dm_exec_requests, dm_exec_sessions, and dm_exec_query_stats, amongst others. This collector type is not configurable and should not be altered other than changing the upload frequency. Therefore, use the Query Statistics data collection set and don't define your collector item using the query activity collector type.

Creating Collection Sets and Items

I am going to create a data collection set with four collection items. To more effectively explain the code, it will be broken into shorter blocks, each accomplishing a different task. The first task will be to start a transaction, use a try block, to declare a couple of variables that you will need, and then to create the data collection set named My Performance DC, as shown here.

```
USE msdb;
BEGIN TRANSACTION

BEGIN TRY
    DECLARE @collection_set_id INT

    EXEC [dbo].[sp_syscollector_create_collection_set]
        @name = N'My Performance DC',
        @collection_mode = 1, -- Non-cached
        @description = N'Collects performance data and row counts for user tables.',
        @days_until_expiration = 7,
        @schedule_name = N'CollectorSchedule_Every_6h',
        @collection_set_id = @collection_set_id OUTPUT
```

You should take note of the @collection_set_id variable: It will hold the integer ID for the collection set that will be used by the stored procedures that add collection items to that collection set. Now that you have that ID, you can create the collection items. But first, you need the globally unique identifier (GUID) for the collector type.

Note Collection sets and collection items can only be created via T-SQL. SQL Profiler can, however, export a trace definition to a collection set definition T-SQL script. You can also script the system collection sets and use the generated T-SQL to learn how to write T-SQL to create your own collection sets and collection items.

This next part of the code will add two collection items using the T-SQL collector type. The first statement retrieves this GUID, which, like the collection set ID, is used by the subsequent procedures that add the collection items.

```
DECLARE @collector_type_uid UNIQUEIDENTIFIER
     = (SELECT TOP 1 collector_type_uid
        FROM [dbo].[syscollector_collector_types]
        WHERE name = N'Generic T-SQL Query Collector Type')

DECLARE @CollItemID int

EXEC [dbo].[sp_syscollector_create_collection_item]
    @name = N'Row Counts',
    @parameters = N'
    <TSQLQueryCollector>
        <Query>
            <Value>
                SELECT
                    S.name AS SchemaName,
                    O.name AS ObjectName,
                    SUM(PS.row_count) AS TotalRows
                FROM sys.dm_db_partition_stats AS PS
                    INNER JOIN sys.objects AS O
                      ON O.object_id = PS.object_id

                        INNER JOIN sys.schemas AS S
                          ON S.schema_id = O.schema_id
                WHERE O.is_ms_shipped = 0
                 AND PS.index_id &lt;= 1
                GROUP BY S.name, O.Name
            </Value>
            <OutputTable>row_countage</OutputTable>
        </Query>
        <Databases UseSystemDatabases = "false" UseUserDatabases = "true" />
    </TSQLQueryCollector>',
    @collection_item_id = @CollItemID OUTPUT,
    @frequency = 5,
    @collection_set_id = @collection_set_id,
    @collector_type_uid = @collector_type_uid
```

This collection item, named Row Counts, will collect row count data for all user tables (based primarily on data from the dm_db_partition_stats DMV).

The @parameters parameter is XML that tells the Data Collector what data to collect, and its schema is determined by the collector type. You can see the XML schemas for the various collector types by querying the syscollector_collector_types table in the msdb database. For the T-SQL collector type, the <Value> element in @parameters is the T-SQL code to run. Although these examples only contain a SELECT statement, the code can contain more complex T-SQL, such as the use of variables, control-of-flow (IF, WHILE, etc.), and so on. The output table is the table name that should be used in the management data warehouse.

The next collection item, named SQL Trace Events, is a SQL trace collector type, and although the @parameters value is again XML, this time it contains the trace events, grouped by event type (or event category) that are to be collected by the Data Collector. Because this collector type is actually creating a server-side trace, you should design SQL trace collector types such that they filter for very specific information so that you don't flood the management data warehouse with an excess of unrelated or irrelevant information.

```
SET @collector_type_uid
     = (SELECT TOP 1 collector_type_uid
        FROM [dbo].[syscollector_collector_types]
        WHERE name = N'Generic SQL Trace Collector Type')

EXEC [dbo].[sp_syscollector_create_collection_item]
    @name = N'SQL Trace Events',
    @parameters=N'
    <SqlTraceCollector>
        <Events>
            <EventType id="11" name="Stored Procedures">
                <Event id="10" name="RPC:Completed"

                       columnslist="1,3,11,35,12,28,13" />
                <Event id="45" name="SP:StmtCompleted"
                       columnslist="1,3,11,35,12,28,13" />
            </EventType>
            <EventType id="13" name="TSQL">
                <Event id="12" name="SQL:BatchCompleted"
                       columnslist="1,3,11,35,12,28,13" />
            </EventType>
        </Events>
        <Filters>
            <Filter columnid="35" columnname="DatabaseName"
                    logical_operator="AND" comparison_operator="LIKE"
                    value="MyDatabase"/>
        </Filters>
    </SqlTraceCollector>',
    @collection_item_id = @CollItemID OUTPUT,
    @frequency = 5,
    @collection_set_id = @collection_set_id,
    @collector_type_uid = @collector_type_uid
```

The last collection item, PerfMon Counters, is a performance counter collector type. The @parameters value is yet again XML, but this time, it contains the list of performance counters that the Data Collector is to collect.

```
    SET @collector_type_uid
         = (SELECT TOP 1 collector_type_uid
            FROM [dbo].[syscollector_collector_types]
            WHERE name = N'Performance Counters Collector Type')

    EXEC [dbo].[sp_syscollector_create_collection_item]
        @name = N'PerfMon Counters',
        @parameters=N'
        <PerformanceCountersCollector>
            <PerformanceCounters Objects="Memory" Counters="Pages/sec" />
            <PerformanceCounters Objects="Network Interface"
                                 Counters="Bytes Total/sec" Instances="*" />
            <PerformanceCounters Objects="Process"
                                 Counters="% Processor Time" Instances="*" />
            <PerformanceCounters Objects="$(INSTANCE):Plan Cache"
                                 Counters="Cache Hit Ratio" Instances="*" />
            <PerformanceCounters Objects="$(INSTANCE):Workload Group Stats"
                                 Counters="CPU usage %" Instances="*" />
        </PerformanceCountersCollector>',
        @collection_item_id = @CollItemID OUTPUT,
        @frequency = 5,
        @collection_set_id = @collection_set_id,
        @collector_type_uid = @collector_type_uid
```

Finally, close the TRY block, and either COMMIT the transaction or, if an exception occurs, re-raise the exception and ROLLBACK the transaction.

```
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;

    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE()
            , @ErrorSeverity INT = ERROR_SEVERITY()
            , @ErrorState INT = ERROR_STATE()
            , @ErrorNumber INT = ERROR_NUMBER()
            , @ErrorLine INT = ERROR_LINE()
            , @ErrorProcedure NVARCHAR(200) = ISNULL(ERROR_PROCEDURE(), '-')

    RAISERROR (14684, @ErrorSeverity, 1 , @ErrorNumber, @ErrorSeverity, @ErrorState,
               @ErrorProcedure, @ErrorLine, @ErrorMessage);
END CATCH;
GO
```

Collecting Data

A quick refresh of the Data Collection node in Object Explorer reveals the new collection set. You could double-click the collection set (or right-click and choose Properties) to see its properties, as shown in Figure 2-10.

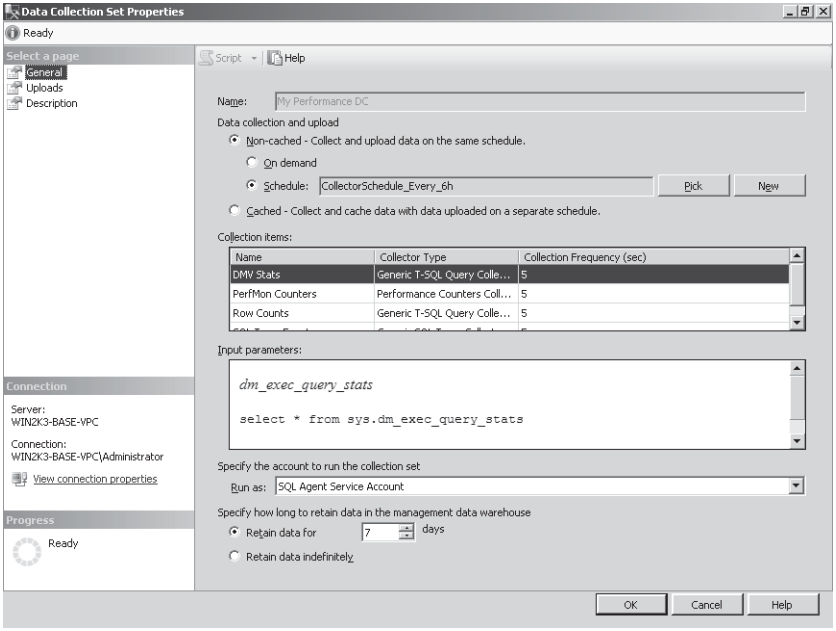


FIGURE 2-10 Data Collection Set Properties dialog box

The collection set does not collect anything, however, until you start the data collection set. This feat is easily accomplished by right-clicking the collection set and choosing Start Data Collection Set, as shown in Figure 2-11.

Note You can also use T-SQL to start a data collection set via the system stored procedure sp_syscollector_start_collection_set.

Once it is started, it will collect and upload data every six hours (as defined in the example data collection set created earlier in this section). You can always manually perform a collection by selecting Collect And Upload Now from the context menu for the data collection set.

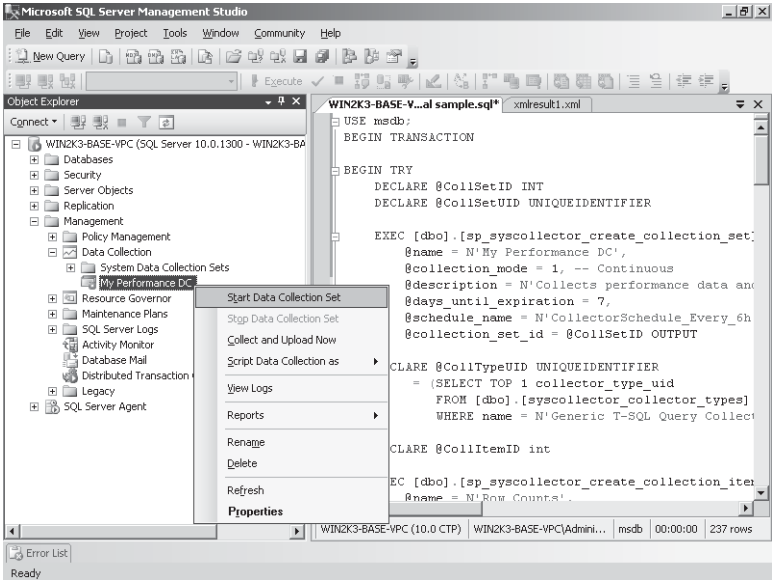


FIGURE 2-11 Starting a data collection set

Viewing Collection Data

Once data collection sets begin to collect and upload data, you can begin to examine the collected data. There are a series of built-in tables, views, and other objects that are created when the management data warehouse is created that are used for the system data collection sets. For example, to view data from the last snapshot for the Disk Usage system collection set, you could query the snapshots.disk_usage table, as shown here.

```
SELECT
     database_name AS DatabaseName
    , dbsize AS DatabaseSize

    , logsize AS LogSize
    , ftsize AS FullTextSize
FROM snapshots.disk_usage
WHERE snapshot_id = (SELECT MAX(snapshot_id) FROM snapshots.disk_usage)
ORDER BY database_name
```

The query would give you results similar to these:

DatabaseName	DatabaseSize	LogSize	FullTextSize
AdventureWorks	29240	2304	0
AdventureWorksDW	9016	256	0
AdventureWorksLT	2688	33024	0
master	512	128	0
model	280	64	0
msdb	2184	192	0
ReportServer	408	800	0
ReportServerTempDB	280	104	0
tempdb	3936	96	0

There are other predefined tables and views for viewing performance data (snapshots.performance_counters view), trace data (snapshots.trace_data table), and results from T-SQL queries (snapshots.os_waits_stats table) from the three system data collection sets. You can find these and others in the management data warehouse database being used by the Data Collector.

The three built-in system data collection sets also each come with a report, which you can view by right-clicking on the system data collection set and choosing Reports, then choosing Historical, and then choosing the specific report you want, such as Disk Usage Summary, as shown in Figure 2-12.

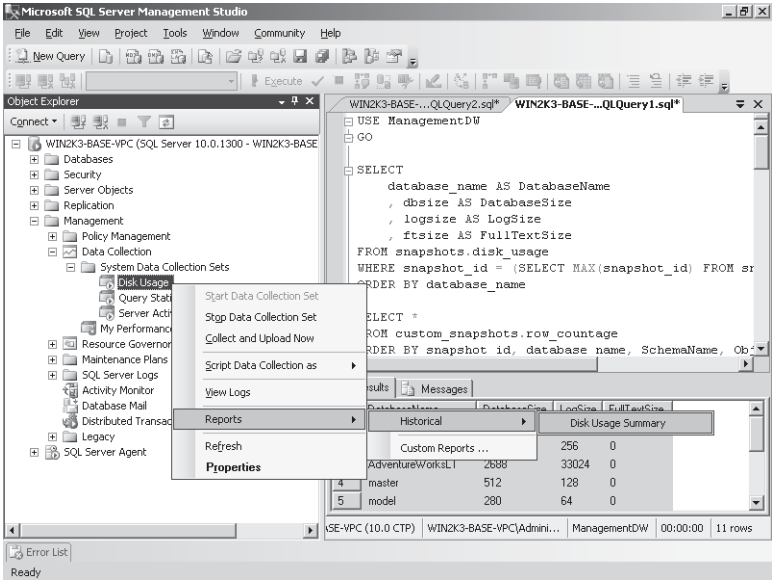


FIGURE 2-12 Data collection reports context menu

Selecting Disk Usage Summary results in the report being displayed in SSMS, as shown in Figure 2-13.

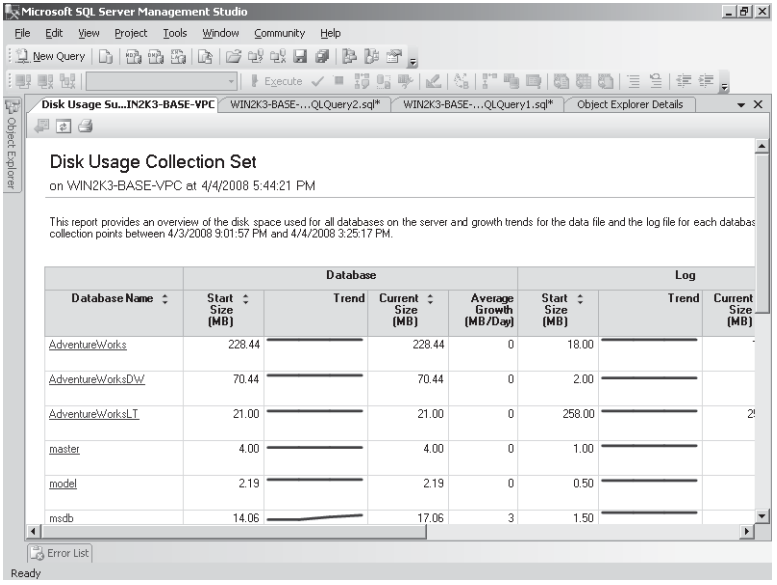


FIGURE 2-13 Disk Usage Collection Set built-in report

In addition to the Disk Usage Summary Report, there are two other built-in reports: Query Statistics History Report and Server Activity History Report. I suggest exploring these reports further as each one not only provides valuable information but has drill-down reports with additional details.

Note Starting with Release Candidate 0, the report context menu items will be changing. Instead of right-clicking the individual system collection set nodes in Object Explorer, you will right-click the Data Collection node. From there, you can choose Reports and then choose Management Data Warehouse, and under that menu item you will find all of the built-in reports.

User Collection Set Data

Performance counter and SQL Trace data from user-created collection sets and items are stored in the same place as their system-defined counterparts. To distinguish the data from the user-created data collection sets and the system data collection sets, you can join to the core.snapshots view and add criteria to limit the GUID of your data collection set. For

example, to view performance counters for the data collection set created earlier in this section, you could execute the following statement.

```
SELECT *
FROM snapshots.performance_counters AS pc
    INNER JOIN core.snapshots AS s ON s.snapshot_id = pc.snapshot_id
WHERE s.collection_set_uid =
      CAST('59ACA7B9-8615-41AF-A634-99ED15B03E56' AS UNIQUEIDENTIFIER)
```

Mind you, the GUID will be different each time you create a collection set (although you can specify a GUID when creating a collection set instead of having a random one assigned).

Now what about the collected T-SQL data? Well, there is a schema in the management data warehouse database named custom_snapshots. When the collection for the user created collection item using a T-SQL collector type is collected and uploaded for the first time, it adds the necessary table to the custom_snapshots schema in the management data warehouse database using the name defined in the collection item (i.e. row_countage and dm_exec_query_stats from the earlier examples).

For example, this following query is using the custom_snapshots.row_countage table specified in the Row Counts collection item defined earlier in this section, and it will report back any changes in row counts from the last two snapshots and any new tables with at least one row that has been created between the last two snapshots.

```
;WITH SnapShotLast AS
(
    SELECT MAX(snapshot_id) AS snapshot_id
    FROM custom_snapshots.row_countage
)
, SnapShotPrevious AS
(
    SELECT MAX(snapshot_id) AS snapshot_id
    FROM custom_snapshots.row_countage
    WHERE snapshot_id < (SELECT TOP 1 snapshot_id FROM SnapShotLast)
)
, r1 as
(
    SELECT *
    FROM custom_snapshots.row_countage
    WHERE snapshot_id = (SELECT TOP 1 snapshot_id FROM SnapShotLast)
)
, r2 AS
(
    SELECT *
    FROM custom_snapshots.row_countage
```

```
    WHERE snapshot_id = (SELECT TOP 1 snapshot_id FROM SnapShotPrevious)
)
SELECT
    r1.database_name
    , r1.SchemaName
    , r1.ObjectName
    , MAX(ABS(r1.TotalRows - ISNULL(r2.TotalRows, 0))) RowDelta
FROM r1
    LEFT OUTER JOIN r2
        ON r1.database_name = r2.database_name
        AND r1.SchemaName = r2.SchemaName
        AND r1.ObjectName = r2.ObjectName
WHERE r1.TotalRows != ISNULL(r2.TotalRows, 0)
GROUP BY r1.database_name, r1.SchemaName, r1.ObjectName
```

This query could be created as a view or stored procedure in the management data warehouse database and be the source for a report.

Query Plan Freezing

SQL Server 2005 introduced a feature known as plan forcing. Plan forcing allows you to specify a query plan to be used when a query is executed. Of course, you would only use plan forcing when the optimizer was making obviously bad plan choices, which is infrequent at best. On that off occasion, however, when you need to give the optimizer a push in the right direction, you could use plan forcing to do so.

SQL Server 2008 takes this concept further with the advent of plan freezing, which allows you to easily use plans already in cache as the basis for your plan guides.

Plan Forcing

There are two ways in which you could force a plan to be used. The first method is using the USE PLAN query hint. This requires an XML plan, which could be obtained from one of several methods, including:

- SET SHOWPLAN XML
- SET STATISTICS XML
- sys.dm_exec_query_plan
- SQL Server Profiler

Once you have a plan, you could then use it in the query with the USE PLAN query hint. This method allows you use a plan on a case-by-case basis and to optionally customize, optimize, or finely tune the plan as needed. Several examples of this can be found in SQL Server Books Online. Here is an abridged example of such usage:

```
SELECT *
FROM Sales.SalesOrderHeader h
    INNER JOIN Sales.Customer c ON h.CustomerID = c.CustomerID
WHERE h.[SalesPersonID] = 279
OPTION (USE PLAN N'<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" Build="9.00.3054.00">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="    SELECT *&#xD;&#xA;    FROM Sales.
SalesOrderHeader h&#xD;&#xA;&#x9;&#x9;INNER JOIN Sales.Customer c ON h.CustomerID =
c.CustomerID&#xD;&#xA;    WHERE h.[SalesPersonID] = 282&#xD;&#xA;" StatementId="1"
StatementCompId="1" StatementType="SELECT" StatementSubTreeCost="0.741371"
StatementEstRows="270.066" StatementOptmLevel="FULL" StatementOptmEarlyAbortReason="Go
odEnoughPlanFound">
          <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true" CONCAT_
NULL_YIELDS_NULL="false" ANSI_NULLS="false" ANSI_PADDING="false" ANSI_WARNINGS="false"
NUMERIC_ROUNDABORT="false" />


          <QueryPlan CachedPlanSize="36" CompileTime="14" CompileCPU="14"
CompileMemory="528">

<!-- The remainder of this plan was removed for space reasons -->

          </QueryPlan>
          <UDF ProcName="[AdventureWorks].[dbo].[ufnLeadingZeros]">
            <Statements>
              <StmtSimple StatementText="&#xD;&#xA;CREATE FUNCTION [dbo].[ufnLeadingZ
eros](&#xD;&#xA;    @Value int&#xD;&#xA;) &#xD;&#xA;RETURNS varchar(8) &#xD;&#xA;WITH
SCHEMABINDING &#xD;&#xA;AS &#xD;&#xA;BEGIN&#xD;&#xA;

    DECLARE @ReturnValue varchar(8);&#xD;&#xA;&#xD;&#xA;
    SET @ReturnValue = CONVERT(varchar(8), @Value);&#xD;&#xA;
   " StatementId="2" StatementCompId="3" StatementType="ASSIGN" />
              <StmtSimple StatementText=" SET @ReturnValue = REPLICATE(''0'', 8 -
DATALENGTH(@ReturnValue)) + @ReturnValue;&#xD;&#xA;&#xD;&#xA;
   " StatementId="3" StatementCompId="4" StatementType="ASSIGN" />
              <StmtSimple StatementText=" RETURN (@ReturnValue);&#xD;" StatementId="4"
StatementCompId="5" StatementType="RETURN" />
            </Statements>
          </UDF>
        </StmtSimple>
      </Statements>
    </Batch>
  </BatchSequence>
</ShowPlanXML>')
```

The second method to implement plan forcing is through the use of the sp_create_plan_guide system stored procedure. Unlike the USE PLAN option, using sp_create_plan_guide is executed once and then applied to all queries matching the statement text. It will be in effect for matching queries until the plan guide is removed or becomes invalid (e.g., because of an index being dropped). Also, using the USE PLAN option requires making changes to the application code, which may be difficult to implement or simply not allowed. Creating plan guides, however, can be implemented without touching the application code and may be your only option in some scenarios.

Here is an example:

```
USE AdventureWorks
GO

CREATE PROCEDURE Sales.SalesBySalesPersonID  (@SalesPersonID INT)
AS
BEGIN
    SELECT *
    FROM Sales.SalesOrderHeader h
        INNER JOIN Sales.Customer c ON h.CustomerID = c.CustomerID
    WHERE h.[SalesPersonID] = @SalesPersonID
END

-- *Turn on “Include Actual Execution Plan” to see the plans

-- The first null value causes a bad plan for subsequent non-null values.
EXEC Sales.SalesBySalesPersonID NULL
EXEC Sales.SalesBySalesPersonID 282
GO

-- Now let’s optimize for a non-null parameter value
EXEC sp_create_plan_guide
    @name = N’SalesBySalesPersonID_Guide’,

    @stmt = N’SELECT *
    FROM Sales.SalesOrderHeader h
        INNER JOIN Sales.Customer c ON h.CustomerID = c.CustomerID
    WHERE h.[SalesPersonID] = @SalesPersonID’,
    @type = N’OBJECT’,
    @module_or_batch = N’Sales.SalesBySalesPersonID’,
    @params = NULL,
    @hints = N’OPTION (OPTIMIZE FOR (@SalesPersonID = 282))’
GO

-- Free the proc cache…
DBCC FREEPROCCACHE
GO

-- Now they will use the plan optimized for a non-null value
EXEC Sales.SalesBySalesPersonID NULL
EXEC Sales.SalesBySalesPersonID 282
GO
```

Note In SQL Server 2008, you can now use plan guides with all data manipulation language (DML) statements, including the new MERGE statement.

Plan Freezing

In SQL Server 2005, you could implement plan forcing using a plan already in cache (i.e., freeze the plan), but the process was not simple. In SQL Server 2008, you can easily create the plan guide from cache using a new system stored procedure named sp_create_plan_guide_from_handle.

```
USE AdventureWorks;
GO

SELECT c.CustomerID, c.AccountNumber, h.OrderDate, h.SalesOrderID, h.TotalDue
FROM Sales.SalesOrderHeader h
    INNER JOIN Sales.Customer c ON h.CustomerID = c.CustomerID
WHERE h.[SalesPersonID] = 282
GO

DECLARE @plan_handle varbinary(64);
DECLARE @offset int;

-- Get the plan handle for the query from the plan in cache
SELECT
    @plan_handle = plan_handle,
    @offset = qs.statement_start_offset
FROM sys.dm_exec_query_stats AS qs

    CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS st
    CROSS APPLY sys.dm_exec_text_query_plan
        (qs.plan_handle, qs.statement_start_offset, qs.statement_end_offset) AS qp
WHERE text LIKE N'SELECT c.CustomerID, c.AccountNumber, h.OrderDate, h.SalesOrderID%';

-- Create the plan guide from the cached plan (using the handle from above)
EXECUTE sp_create_plan_guide_from_handle
    @name =  N'CustomerSalesGuide',
    @plan_handle = @plan_handle,
    @statement_start_offset = @offset;
GO
```

You can now also validate the plan using the new sys.fn_validate_plan_guide function. So, if any metadata changes have been made in the database, you can check to see if any plan guides have become invalid. Continuing the example from above:

```
DROP INDEX [IX_Customer_TerritoryID] ON [Sales].[Customer] WITH ( ONLINE = OFF )
GO

SELECT plan_guide_id, msgnum, severity, state, message
FROM sys.plan_guides
CROSS APPLY fn_validate_plan_guide(plan_guide_id);
```

This query would cause the plan guide created in the previous example to become invalid. An easy way to ensure this check is done on each change to metadata is to use a data definition language (DDL) trigger, as follows:

```
USE [AdventureWorks]
GO

CREATE TRIGGER [trddl_PlanValidate] ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS
        (SELECT plan_guide_id, msgnum, severity, state, message
        FROM sys.plan_guides CROSS APPLY sys.fn_validate_plan_guide(plan_guide_id))
      AND EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(max)')
          NOT LIKE '%OVERRIDE_PLAN_GUIDE%'
     BEGIN
        ROLLBACK TRANSACTION;
        RAISERROR('The change you are attempting will invalidate a existing plan
guide.
In order to implement this changes, you can:
1. Remove the offending plan guide
2. Disable this database trigger
3. Include OVERRIDE_PLAN_GUIDE in a comment in your code', 15, 1)

     END
END;
GO
```

So if you then executed this code again:

```
DROP INDEX [IX_Customer_TerritoryID] ON [Sales].[Customer] WITH ( ONLINE = OFF )
GO
```

You'd get this error:

```
Msg 50000, Level 15, State 1, Procedure trddl_PlanValidate, Line 14
The change you are attempting will invalidate a existing plan guide.
In order to implement this changes, you can:
1. Remove the offending plan guide
2. Disable this database trigger
3. Include OVERRIDE_PLAN_GUIDE in a comment in your code
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

But the trigger was designed to allow you to override it by including OVERRIDE_PLAN_GUIDE in a comment, as shown here:

```
DROP INDEX [IX_Customer_TerritoryID] ON [Sales].[Customer] WITH ( ONLINE = OFF )
--OVERRIDE_PLAN_GUIDE
GO
```

Of course, you could simply code the trigger to warn you about the invalid plan guide but not prevent the change. The options are limitless, really.

Note In SQL Server 2008, an invalid plan guide does not cause a query to fail. Instead, the plan is compiled without using the plan guide, which is not preferred for performance reasons.

Another cool trick that people have been asking about: freezing a set from the plan cache using a cursor. Here is an example:

```
-- DBA loops over the plans he/she wants to freeze
declare @plan_handle varbinary(64)
declare @offset int
declare @pgname nvarchar(200)

declare cur_qstats cursor
for select qs.plan_handle, qs.statement_start_offset
from sys.dm_exec_query_stats qs
cross apply sys.dm_exec_sql_text(sql_handle) st
where text LIKE N'select * from t1 inner join t2 on t1.a = t2.a where t1.b > ''5''%'

open cur_qstats
fetch next from cur_qstats
into @plan_handle, @offset

while @@FETCH_STATUS = 0
begin
    set @pgname = N'Guide1' + convert(nvarchar(50), newid())
```

```
    exec sp_create_plan_guide_from_handle
        @name =  @pgname,
        @plan_handle = @plan_handle,
        @statement_start_offset = @offset

    fetch next from cur_qstats
    into @plan_handle, @offset
end

close cur_qstats
deallocate cur_qstats
go
```

Viewing Plan Guides

Now that you have created these plan guides, how do you go about viewing them? You could always use T-SQL to query the system catalog, such as:

```
SELECT *
FROM sys.plan_guides
ORDER BY [name]
```

And now you can also use Object Explorer in SSMS to see a list of plans and their associated details, as shown in Figure 2-14.

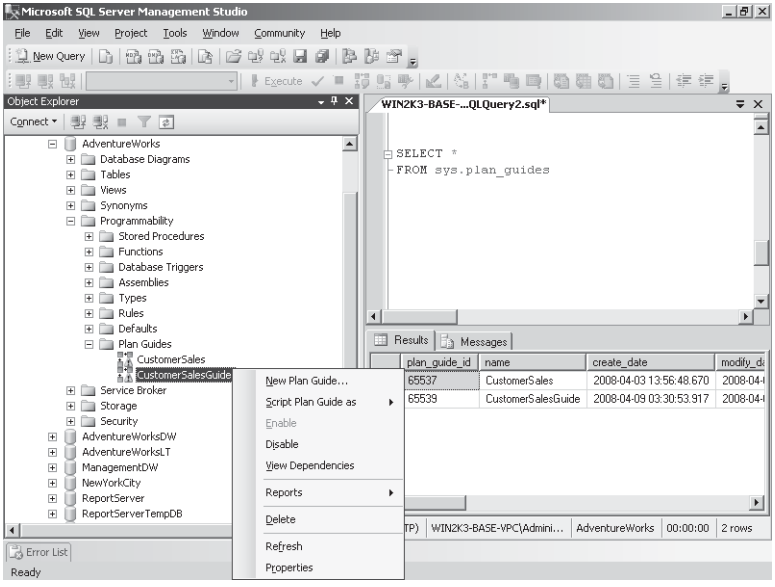


FIGURE 2-14 Plan guide context menu in Object Explorer

Plan guides are located under the Programmability node of a database in Object Explorer. You can right-click on the individual plan guides (as shown in Figure 2-14) or on the Plan Guides node itself. These context menu options allow you to perform a variety of actions, such as:

- Create a new plan guide
- Script a plan guide
- Enable or disable individual or all plan guides
- Delete a plan guide

You can also view the properties on an individual plan guide, as shown in Figure 2-15.

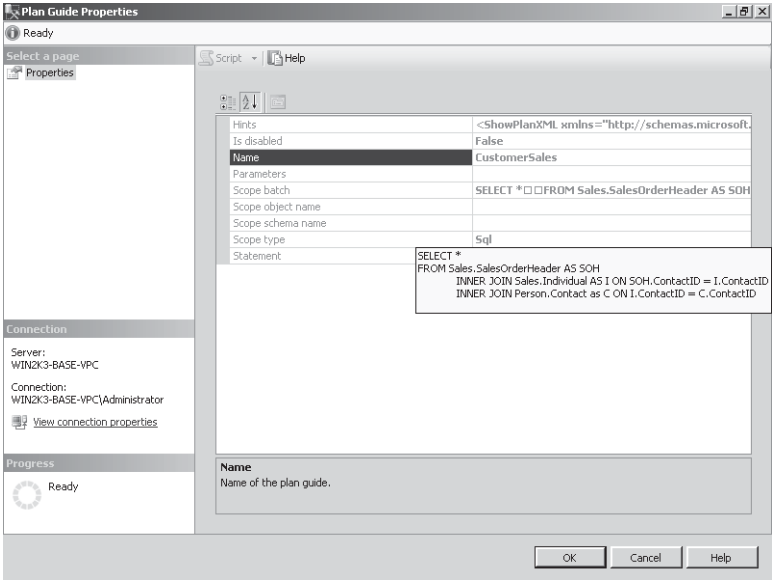


FIGURE 2-15 Plan Guide properties

You can easily view a property value by hovering the mouse cursor over the individual property, or you can select and copy the property value and paste it to a new query window.

You can also see plan guide use via SQL Server Profiler. Under the Performance event category (shown in Figure 2-16), you will find two new events:

- **Plan Guide Successful** This event occurs when SQL Server successfully produces an execution plan for a query or batch with a plan guide.

- **Plan Guide Unsuccessful** This event occurs when SQL Server is unable to produce an execution plan for a query or batch with a plan guide. This event can be caused by an invalid plan guide.

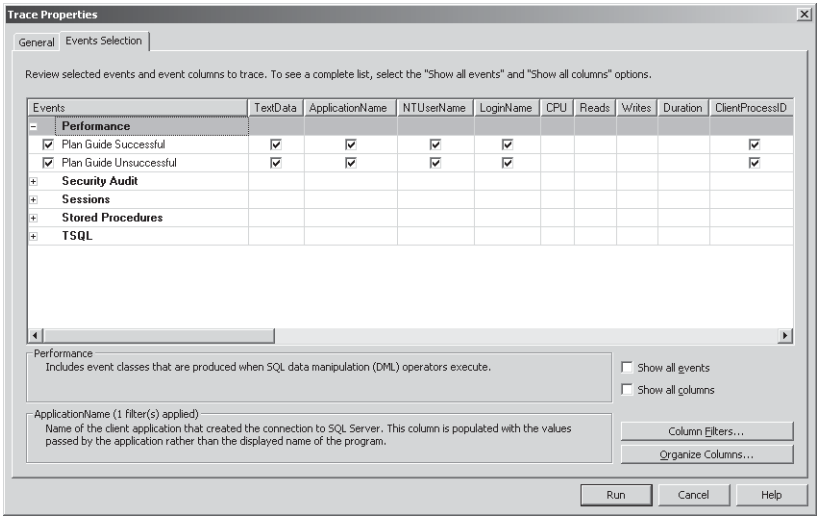


FIGURE 2-16 Trace Properties showing plan guide events

Summary

A poorly designed and implemented database can hurt performance as the database scales up in size. But some well-designed databases can still succumb to performance woes. Fortunately these new performance-related features in SQL Server 2008 can help you:

- More easily monitor performance using the data collection feature.
- Tune your plan guide usage to increase query performance.
- Distribute application workloads via Resource Governor.
- Decrease the performance impact of disk I/O by implementing data and backup compression.

So if you do encounter problems, you can more easily find those problems, and, using these new features, you can optimize the performance of your database server.

Chapter 3

Type System

Introduction

What happens when a relational database management system (RDBMS) starts to manage data that is not so relational? Can an RDBMS go "beyond relational"? SQL Server 2005 showed us that, indeed, a database server can extend beyond the realm of relational data by including the ability to manage semi-structured Extensible Markup Language (*XML*) data. SQL Server 2008 takes this "beyond relational" concept further still by extending the type system yet again.

SQL Server 2008 has made some significant changes to the data type system. Some of these changes are extensions of the existing types and fit into the concept of a relational model, such as the new, higher-precision date and time data types. Some of them go much further. The *XML* data type has been enhanced. A new data type for managing hierarchical data, HIERARCHYID, has been added. New spatial data types, GEOGRAPHY and GEOMETRY, have been added for managing geospatial and geometric data. And SQL Server 2008 can now store binary large objects (BLOBs) as files in the file system instead of its data files, and it can even provide or limit access to these files.

HIERARCHYID

Back in the days of SQL Server 2005, I had been working on a more efficient means of storing hierarchical data in a SQL Server. I had read the gamut of topics, from the adjacency model to nested sets to nested intervals with Farey fractions to materialized paths. I choose the last path (no pun intended) of materialized paths, because I realized it was, in many ways, very efficient and that I could implement a nifty common language runtime (CLR)-based user-defined type (UDT) that would take care of a majority of the functionality needed.

HIERARCHYID uses the same underlying concept, materialized paths, to allow you to efficiently manage hierarchical data. And believe it or not, it is a CLR-based type. However, unlike a UDT (which is created by users, deployed to databases as needed, and requires the CLR Enabled configuration to be turned on), HIERARCHYID is a system CLR type, is built into the SqlTypes library (SqlHierarchyId), and has no dependencies on the CLR Enabled option.

Before I go digging into the coding aspects, I'd like to first give some background information about this new data type.

Compact Design

Unlike my materialized path solution (and other materialized path solutions I have seen), HIERARCHYID offers a compact form of the actual path. In many of the other solutions, the path requires 4 bytes or more per level in the path, which is far less compact than HIERARCHYID. How much space HIERARCHYID actually requires depends on the total number of nodes and the average fanout, or number of children per node. For a fanout of 7 or less, the formula $6*\log_A n$ can be used to determine the maximum number of bits required for any given node in the hierarchy (although most will require less), where *A* represents the average number of children per level, and *n* represents the total number of nodes. The following chart shows a comparison of the maximum possible number of bytes per node required by the two methodologies:

Approximate Node Count	Average Fanout	Approximate Total Levels	HIERARCHYID Maximum Byte Size	Materialized Path Maximum Byte Size
10,000	6	7	4	28
10,000	3	10	7	40
100,000	6	8	5	32
100,000	3	12	8	48
1,000,000	6	9	6	36
1,000,000	3	14	10	56
10,000,000	6	10	7	40
10,000,000	3	16	12	64

Considering that in an evenly distributed hierarchy, a majority of the nodes will be at the deepest level, this means that a significantly smaller amount of space is needed for HIERARCHYID. For example, in a hierarchy that has approximately 1,000,000 nodes and an average fanout of 3 nodes and having about 600,000 nodes at the deepest level, for this level, HIERARCHYID would require at most only 10 bytes per node, for a maximum total of 6,000,000 bytes, or less than 6 megabytes (MB) for those nodes. The other materialized path model would require 56 bytes per node at the deepest level for a total of 33,600,000 bytes, or just more than 32 MB—a difference of more than 26 MB. For all of the data, a materialized path would require approximately 45 MB of space, and HIERARCHYID would require a maximum of 8 MB.

Creating and Managing a Hierarchy

HIERARCHYID is very flexible and easy to use. Although it is simple to implement, if used incorrectly, it will not maintain a valid hierarchy. This section will show you how to properly implement and maintain a HIERARCHYID column.

The Root Node

The first step to any hierarchy is to establish a root node of the hierarchy, which is done using the static GetRoot method of the HIERARCHYID data type, as shown here:

```
CREATE TABLE [dbo].[Folder]
(
    [FolderNode] HIERARCHYID NOT NULL
    [FolderID] INT NOT NULL,
    [ParentFolderID] INT NULL,
    [Description] NVARCHAR(50) NULL,
    CONSTRAINT [PK_Folder_FolderNode]
        PRIMARY KEY CLUSTERED ([FolderNode] ASC)
);
GO

INSERT INTO [dbo].[Folder]
VALUES (HIERARCHYID::GetRoot(), 1, NULL, 'A')
```

GetRoot returns a zero-length HIERARCHYID node. If the HIERARCHYID column is the only thing that uniquely identifies the nodes of the hierarchy, then you can only have one root node. You can have multiple root nodes if you have a compound key field.

```
CREATE TABLE [dbo].[Folder]
(
    [Drive] CHAR(1) NOT NULL,
    [FolderNode] HIERARCHYID NOT NULL
    [FolderID] INT NOT NULL,
    [ParentFolderID] INT NULL,
    [Description] NVARCHAR(50) NULL,
    CONSTRAINT [PK_Folder_DriveFolderNode]
        PRIMARY KEY CLUSTERED ([Drive] ASC, [FolderNode] ASC)
);
GO

INSERT INTO [dbo].[Folder]
VALUES
    ('C', HIERARCHYID::GetRoot(), 1, NULL, 'A')
    ('D', HIERARCHYID::GetRoot(), 2, NULL, 'A')
```

In the preceding example, each unique "drive" can have its own separate hierarchy, each with a root node, because the Drive and FolderNode form a composite key for the table.

Adding Child Nodes

Once you have a root, you can then proceed to add children to the hierarchy. There are two methods for adding a child node: The first is to use the GetDescendant method, and the

second is to manually construct a new path. I will address the latter in a later section of this chapter. In the meantime, let's delve into the former.

The GetDescendant method allows you to generate a child node that:

- Is the first child node.
- Comes after an existing node.
- Comes before an existing node.
- Comes in between two existing nodes.

To achieve this functionality, the GetDescendant method has two nullable parameters that determine which type of node will be generated. For example, to generate the first child node, you would do the following:

```
DECLARE @folderNode HIERARCHYID;

SELECT @folderNode = FolderNode
FROM [dbo].[Folder]
WHERE FolderID = 2;

DECLARE @newChildFolderNode HIERARCHYID = @folderNode.GetDescendant(NULL, NULL);

SELECT
    @folderNode AS FolderNode,
    @folderNode.ToString() AS FolderNodePath,
    @newChildFolderNode AS ChildFolderNode,
    @newChildFolderNode.ToString() AS ChildFolderNodePath;
```

This would give the following results:

FolderNode	FolderNodePath	ChildFolderNode	ChildFolderNodePath
0x58	/1/	0x5AC0	/1/1/

The child path appends "1/" onto the existing parent path. As a matter of fact, whenever you create the first child node (using NULL for both parameters) under a given node, it will always append "1/" to the existing parent path. The GetDescendant method is, in fact, deterministic.

The following table shows how all four variations work:

Parameter 1	Parameter 2	Description
NULL	NULL	New (first) child node
Child Node	NULL	Child node that comes after the child node of parameter 1
NULL	Child Node	Child node that comes before the child node of parameter 2
Child Node	Child Node	Child node that comes between the child nodes

To demonstrate how these work, let's create the hierarchy of folders shown in Figure 3-1.

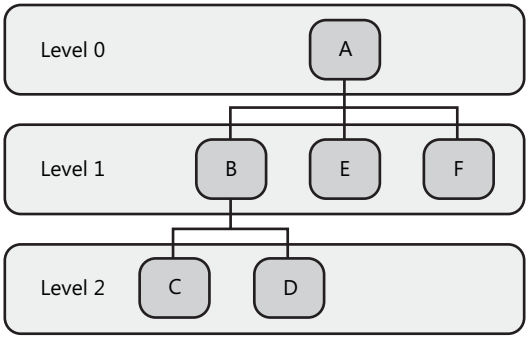


FIGURE 3-1 A simple hierarchy

We will use a simplified version of the Folder table in this example.

```
CREATE TABLE [dbo].[Folder]
(
    [FolderNode] HIERARCHYID NOT NULL,
    [Description] NVARCHAR(50) NOT NULL
);
GO
--Add node A
DECLARE @folderNode HIERARCHYID = HIERARCHYID::GetRoot();

INSERT INTO [dbo].[Folder]
VALUES (@folderNode, 'A')

--Add node B (first child, so NULL, NULL)
DECLARE @childFolderNode HIERARCHYID = @folderNode.GetDescendant(NULL, NULL);

INSERT INTO [dbo].[Folder]
VALUES (@childFolderNode, 'B')

--Add node E (second child, so Child 1, NULL)
SET @childFolderNode = @folderNode.GetDescendant(@childFolderNode, NULL);

INSERT INTO [dbo].[Folder]
VALUES (@childFolderNode, 'E')

--Add node F (third child, so Child 2, NULL)
SET @childFolderNode = @folderNode.GetDescendant(@childFolderNode, NULL);

INSERT INTO [dbo].[Folder]
VALUES (@childFolderNode, 'F')

SELECT @folderNode = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'B'
```

```
--Add node C (first child, so NULL, NULL)
SET @childFolderNode = @folderNode.GetDescendant(NULL, NULL);

INSERT INTO [dbo].[Folder]
VALUES (@childFolderNode, 'C')

--Add node D (second child, so Child 1, NULL)
SET @childFolderNode = @folderNode.GetDescendant(@childFolderNode, NULL);

INSERT INTO [dbo].[Folder]
VALUES (@childFolderNode, 'D')
```

This may at first seem like a lot of work for as simple a hierarchy that is being exemplified, but really it's just a simple process to add a node, and you would likely write a stored procedure to do the work. Also, you normally wouldn't add nodes in this fashion. You would more likely add a set of nodes based on some existing hierarchical data (using some other hierarchy model) that could be inserted using a single INSERT statement with some common table expressions (CTEs), or pass the structure in using XML and insert.

If I did want to add a node C1 between nodes C and D, here is how it could be done:

```
DECLARE @folderNode HIERARCHYID;
DECLARE @childfolderNode1 HIERARCHYID;
DECLARE @childfolderNode2 HIERARCHYID;

SELECT @folderNode = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'B' --existing path /1/

SELECT @childfolderNode1 = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'C' --existing path /1/1/

SELECT @childfolderNode2 = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'D' --existing path /1/2/

--Add node C1 (middle child, so Child 1, Child 2)
SET @folderNode = @folderNode.GetDescendant(@childfolderNode1, @childfolderNode2);

INSERT INTO [dbo].[Folder]
VALUES (@folderNode, 'C1') --new path will be /1/1.1/
```

Notice how we need to know the two child nodes as well as the parent node under which the new child is being added. In the coming section, we will see how this can be somewhat simplified by using some of the other methods of the HIERARCHYID data type.

Querying

We have already seen two of the HIERARCHYID methods already: GetRoot and GetDescendant. This new data type has several additional methods that allow you to determine things such as at what level the node is located; what node is the direct parent, grandparent, and so on; and even if one node is a descendant of another node.

Levels and Paths

The GetLevel method does exactly what its name states: gets the level of the node. The method takes no parameters and returns a SMALLINT, which represents the level of the node in question.

The ToString method returns a string representation of the path.

```
SELECT *, FolderNode.GetLevel() AS Level, FolderNode.ToString() As Path
FROM Folder;
```

This query produces the following results:

FolderNode	Description	Level	Path
0x	A	0	/
0x58	B	1	/1/
0x68	E	1	/2/
0x78	F	1	/3/
0x5AC0	C	2	/1/1/
0x5B40	D	2	/1/2/

The string path can be used to create a new instance of a HIERARCHYID. The static *Parse* method (which is used for implicit conversion from a string) takes the string path as input and converts it to a *HIERACHYID*, as shown here.

```
DECLARE @h1 HIERARCHYID = HIERARCHYID::Parse('/1/1/');
DECLARE @h2 HIERARCHYID = '/1/1/';
```

These two statements are functionally the same, and both of them create a node equivalent to the FolderNode 0x5AC0 in the table above. Later in this section, this feature will be used to generate HIERARCHYID values for an entire table in a single statement.

Relationships

The *IsDescendant* accepts a single parameter of a HIERARCHYID data type and returns a true value if that HIERARCHYID is a direct or indirect descendant of the node calling the method.

For example, the following script would return all of the descendants of node B (including node B):

```
DECLARE @B HIERARCHYID;

SELECT @B = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'B';

SELECT *, FolderNode.ToString() AS Path
FROM [dbo].[Folder]
WHERE @B.IsDescendant(FolderNode) = 1;
```

To have the query not return node B, simply exclude it as follows:

```
SELECT *, FolderNode.ToString() AS Path
FROM [dbo].[Folder]
WHERE @B.IsDescendant(FolderNode) = 1
 AND FolderNode > @B;
```

The *GetAncestor* method returns a node that is *n* levels higher than the node calling the method. For example, if you want to find out what is the direct parent of node C:

```
DECLARE @C HIERARCHYID;

SELECT @C = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'C';

SELECT *, FolderNode.ToString() AS Path
FROM [dbo].[Folder]
WHERE FolderNode = @C.GetAncestor(1);
```

A node's parent is always supposed to be unique, but because a node can have multiple children, this method does not allow a negative value to be passed in as the parameter. You can, however, pass a value of zero, which is equivalent to the calling node.

The *Reparent* method does exactly that—it changes the parent of a node. It is important to note that if you reparent a node that also has any subordinates, then you need to reparent all of the subordinates as well in order to maintain data integrity of your hierarchy (unless you are swapping two parents or allow orphaned nodes, of course). The following shows a simple example of moving the node C from its existing parent to a new parent of node E.

```
DECLARE @E HIERARCHYID;

SELECT @E = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'E';

UPDATE [dbo].[Folder]
SET FolderNode = FolderNode.Reparent(FolderNode.GetAncestor(1), @E)
WHERE Description = 'C';
```

Moving a section of a tree seems like it would be a little more daunting, but the simplicity of such an operation can only be stated in code:

```
DECLARE @B HIERARCHYID, @E HIERARCHYID,
    @E1 HIERARCHYID, @E2 HIERARCHYID;

SELECT *, FolderNode.ToString() AS Path
FROM [dbo].[Folder]
ORDER BY FolderNode

SELECT @E = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'E'

SELECT @E1 = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'E1'

SELECT @B = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'B'

INSERT INTO [dbo].[Folder]
VALUES
(
  @E1.GetAncestor(1).GetDescendant(@E1, NULL)
  , 'E2'
)

SELECT @E2 = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'E2'

UPDATE [dbo].[Folder]
SET FolderNode = FolderNode.Reparent(@B.GetAncestor(1), @E2)
WHERE @B.IsDescendant(FolderNode) = 1
 AND FolderNode != @B

DELETE [dbo].[Folder]
```

```
WHERE Description = 'E2'

UPDATE [dbo].[Folder]
SET FolderNode = @E2
WHERE FolderNode = @B
```

The beauty of the *Reparent* method is that it can be used to move a child node, grandchild node, and so on. However, when the new parent already has children, the process goes as follows:

1. Create a new node (E2) under the new parent (E).
2. Reparent all the subordinate nodes of B, but not B itself, under the new E2 node.
3. Remove E2.
4. Update B to be E2 by changing its HIERARCHYID to the HIERARCHYID of E2.

Please note that if E had no other children, you could simply reparent B and all of its subordinates.

In this case, all subordinates of node B are moved from under node B to under node E2. All the subordinates are updated in a single statement, but B requires a little more work to keep the paths unique.

Final Thoughts

Earlier in this section, we saw some code that added a child node between two other nodes. Here is the same functionality rewritten in a more concise manner.

```
DECLARE @C HIERARCHYID, @D HIERARCHYID;

SELECT @C = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'C'

SELECT @D = FolderNode
FROM [dbo].[Folder]
WHERE Description = 'D'

INSERT INTO [dbo].[Folder]
VALUES
(
  @C.GetAncestor(1).GetDescendant(@C, @D)
  , 'C1'
)
```

In this example, @C represents node C and @D represents node D. We don't need to prefetch the parent of nodes C and D because we can use the *GetAncestor* method to get that parent.

Indexing

There are two main indexing strategies that you can use for hierarchical data: depth first and breadth first. A depth-first index would be an index on the HIERARCHYID column. The underlying storage of HIERARCHYID is not only compact but also in a depth-first order. Figure 3-2 shows an example of the order of nodes in a depth-first index.

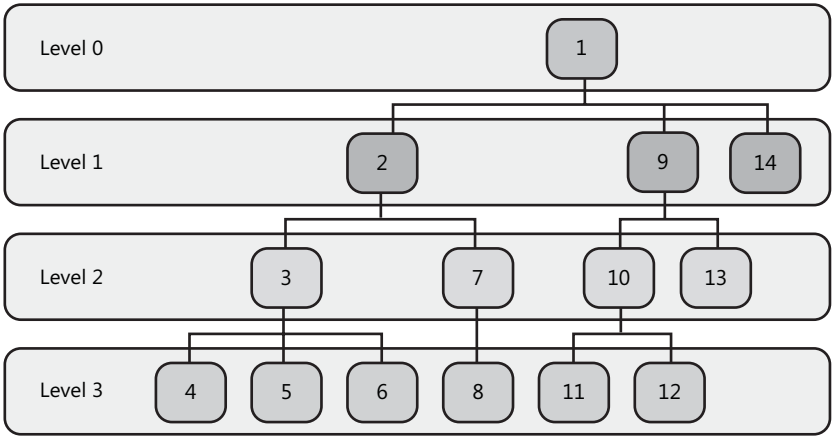


FIGURE 3-2 Depth-first indexing

A depth-first index is best used when you are frequently querying for a node and all of its subordinate nodes, because they are stored closer to each other in this indexing scheme. For example, if you regularly search for a folder and its subfolders, you are best served with a depth-first index. In Figure 3-2, all of the subordinate nodes of node 2 are between node 2 and the sibling of node 2, node 9. All the subordinates of node 3 are between node 3 and node the sibling of 3, node 7.

Note When querying for nodes and subordinates near the top of the hierarchy, there will be a larger number of subordinate nodes and, thus, less likelihood that a depth-first index will be used.

A breadth-first index is best used when you frequently want to query for the direct subordinates of a node, because all the nodes at a given level are always near each other in the index. Figure 3-3 shows an example of node order in a breadth-first index.

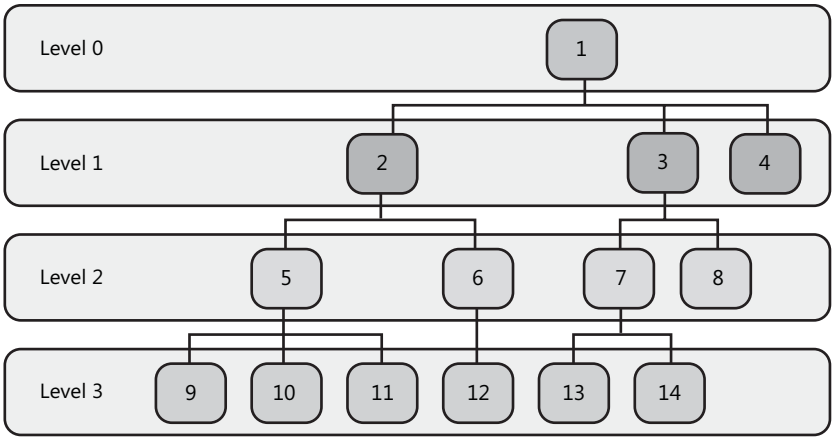


FIGURE 3-3 Breadth-first indexing

In this indexing scheme, all the nodes at the same level are always adjacent to each other. This indexing scheme is more efficient when the hierarchy has a smaller average fanout and many levels. As the average fanout grows in number, the efficiency of a breadth-first index increases.

For example, the nodes at level 2 are nodes 5 through 8, and all of the direct subordinate nodes of node 3 (level 1) are level 2 nodes 7 and 8. When node 3 has many more subordinates, using a breadth-first index will speed up access to these direct subordinate nodes.

Note When querying for nodes and subordinates deep in the hierarchy, there will be a larger number of nodes in the level and, therefore, less likelihood that a breadth-first index will be used, particularly on the deepest level.

What would these two indexes look like in Transact-SQL (T-SQL)? Well, a depth-first is simply an index on a HIERARCHYID column, as shown here:

```
CREATE UNIQUE INDEX inFolder_FolderNode
ON [dbo].[Folder] (FolderNode) ;
```

A breadth-first index is actually implemented as a composite index of the persisted value of the *GetLevel* method and the HIERARCHYID itself. To implement this type of indexing scheme, you need to have a computed column that has the level of the node. The following script shows the new table definition as well as the breadth-first index implementation:

```
CREATE TABLE [dbo].[Folder]
(
```

```
    [FolderNode] HIERARCHYID NOT NULL,
    [Level] AS [FolderNode].GetLevel(), -- PERSISTED? Not in this situation
    [Description] NVARCHAR(50) NOT NULL
);

CREATE UNIQUE INDEX inFolder_LevelFolderNode
ON [dbo].[Folder] (Level, FolderNode);
```

When a UDT method is used as a computed column that needs to be part of an index, you need to specify that it is PERSISTED. System CLR types, however, have no such requirement. So if the breadth-first index is not the clustered index, then you should not persist the *GetLevel* method computed column because it will be more storage efficient and will perform better by not persisting it.

Limitations and Cautions

Even though this new means of managing hierarchical data is versatile and functional, it does have some limitations of which you need to be aware. The most important item of note in this regard is that it is not self-managing. Just because you add a HIERARCHYID column to your table doesn't mean that it actually has any hierarchy information, or even correct hierarchy information, for that matter. It is just another data type, and the data needs to be managed like any other column's data. Assuming that a HIERARCHYID column is self-managed is the same as assuming the ManagerID column is self-managed.

Now that you know that you need to manage the data in a HIERARCHYID column, I must also point out that a HIERARCHYID column is not, by default, a unique column. To ensure you have good data, you will need to use a primary key, a unique constraint, a trigger, or some other mechanism.

And even if you have a unique set of HIERARCHYID values, there is no guarantee that you have a valid hierarchy. You also need to maintain the relationships in your data. For example, let's say you created a table called Folder, as shown here:

```
CREATE TABLE [dbo].[Folder](
    [FolderID] INT NOT NULL,
    [ParentFolderID] INT NULL,
    [Description] NVARCHAR(50) NULL
);
GO
```

That was populated as such:

```
INSERT INTO [dbo].[Folder]
```

```
VALUES
    (1, NULL, 'A'),
    (2, 1, 'B'),
    (3, 2, 'C'),
    (4, 2, 'D'),
    (5, 1, 'E');
```

And then you decided to get add a HIERARCHYID column:

```
ALTER TABLE [dbo].[Folder]
ADD [FolderNode] HIERARCHYID;
```

Now you have a HIERARCHYID column that needs to be populated, so you begin by updating the root node:

```
UPDATE [dbo].[Folder]
SET FolderNode = HIERARCHYID::GetRoot()
WHERE ParentFolderID IS NULL
```

And then you update the next level (those nodes immediately under the root node):

```
UPDATE [dbo].[Folder]
SET FolderNode = HIERARCHYID::GetRoot().GetDescendant(NULL, NULL)
WHERE ParentFolderID = 1
```

You do a quick check of the data by executing SELECT * FROM [dbo].[Folder] to reveal the following data:

FolderID	ParentFolderID	Description	FolderNode
1	NULL	A	0x
2	1	B	0x58
3	2	C	NULL
4	2	D	NULL
5	1	E	0x58

What went wrong? Two of the child folders have the same FolderNode value. The query updated all the children of the root node to the same value. This means that to do a mass update of HIERARCHYIDs, you need to do one of the following:

- Update each node in the hierarchy one at a time using *GetDescendant*.
- Generate paths for all of the nodes in a single query and convert to HIERARCHYID.

In the next section, we will begin some practical applications of using HIERARCHYID, including a method for the latter of these two methods for updating your existing data to use HIERARCHYID.

Working with HIERARCHYID

To better understand how HIERARCHYID really works, we are going to take an existing set of data that is modeled using the adjacency methodology and convert it to use HIERARCHYID.

Converting Adjacency Model to HIERARCHYID

The *AdventureWorks* sample database has a table of employees with a self reference to their respective managers. The first step is to create a table that will hold the HIERARCHYID-based hierarchical representation of the employees.

```
USE [AdventureWorks]
GO

CREATE TABLE [HumanResources].[EmployeeHierarchy](
    EmployeeNode HIERARCHYID NOT NULL,
    EmployeeLevel AS EmployeeNode.GetLevel() PERSISTED,
    [EmployeeID] [int] NOT NULL,
    [ManagerID] [int] NULL,
    [FirstName] [nvarchar](50) NOT NULL,
    [LastName] [nvarchar](50) NOT NULL,
    [Title] [nvarchar](50) NOT NULL,
    CONSTRAINT [PK_EmployeeHierarchy_EmployeeNode]
        PRIMARY KEY CLUSTERED ([EmployeeNode] ASC)
);
GO



CREATE UNIQUE INDEX inEmployeeHierarchy_LevelNode
ON HumanResources.EmployeeHierarchy(EmployeeLevel, EmployeeNode) ;
GO
```

Note You should note that persisting the *ManagerID* field is redundant, because you can use EmployeeNode.GetAncestor(1) to get the same information in just as an efficient manner.

Next we are going to answer a question from the previous section. How do we generate all of the HIERARCHYID values in a single statement (as opposed to doing it row by row)? The answer is to generate the paths by using a combination of two CTEs and the ROW_NUMBER

function. Although the following code is somewhat complex, I want you to first read through the code and see if you can figure out how it is working.

```
;WITH EH (EmployeeID, ManagerID, FirstName, LastName, Title, Position)
AS
(
    SELECT
        E.EmployeeID,
        E.ManagerID,
        C.FirstName,
        C.LastName,
        E.Title,
        ROW_NUMBER() OVER (PARTITION BY ManagerID ORDER BY EmployeeID)
    FROM HumanResources.Employee AS E
        INNER JOIN Person.Contact AS C ON C.ContactID = E.ContactID
)
, EHTemp (EmployeeNode, EmployeeID, ManagerID, FirstName, LastName, Title)
AS
(
    SELECT HIERARCHYID::GetRoot(),
        EmployeeID, ManagerID, FirstName, LastName, Title
    FROM EH
    WHERE ManagerID IS NULL

    UNION ALL

    SELECT
        CAST(EHTemp.EmployeeNode.ToString()
            + CAST(EH.Position AS VARCHAR(30)) + '/' AS HIERARCHYID),
        EH.EmployeeID, EH.ManagerID, EH.FirstName, EH.LastName, EH.Title
    FROM EH
        INNER JOIN EHTemp ON EH.ManagerID = EHTemp.EmployeeID
)
INSERT INTO [HumanResources].[EmployeeHierarchy]
    (EmployeeNode,EmployeeID, ManagerID, FirstName, LastName, Title)
SELECT EmployeeNode, EmployeeID, ManagerID, FirstName, LastName, Title
FROM EHTemp
ORDER BY EmployeeNode;
```

The secret to this solution is twofold. The first CTE creates row numbers for each set of children by using the partitioning ability of the ROW_NUMBER function. The following query is similar to that used in the first CTE.

```
    SELECT
        E.EmployeeID AS EmpID,
        E.ManagerID AS MgrID,
        C.LastName,
        ROW_NUMBER() OVER (PARTITION BY ManagerID ORDER BY EmployeeID) AS RowNum
    FROM HumanResources.Employee AS E
        INNER JOIN Person.Contact AS C ON C.ContactID = E.ContactID
```

When executed, this query returns the following (abridged) results (formatted so the hierarchy is more easily read):

EmpID	MgrID	LastName	RowNum
109	NULL	Sánchez	1
6	109	Bradley	1
2	6	Brown	1
46	6	Harnpadoungsataya	2
106	6	Gibson	3
119	6	Williams	4
203	6	Eminhizer	5
269	6	Benshoof	6
271	6	Wood	7
272	6	Dempsey	8
12	109	Duffy	2
3	12	Tamburello	1
4	3	Walters	1
9	3	Erickson	2
11	3	Goldberg	3
158	3	Miller	4
79	158	Margheim	1
114	158	Matthew	2
217	158	Raheem	3
263	3	Cracium	5
5	263	D'Hers	1
265	263	Galvin	2
267	3	Sullivan	6
270	3	Salavaria	7
Etc.	Etc.	Etc.	Etc.

What is happening is that instead of generating a row number for all the rows, row numbers are generated for each group of children for a given parent. So all the subordinates of Employee Tamburello (ID 3) have a row number from 1 to 4. Children of the folder with an ID of 1 (folders 2 and 5 in this case) will have their own set of row numbers.

This partitioned row numbering alone doesn't solve the problem at hand. You need to combine the partitioned row numbers with the ability to parse a string representation of the path into a HIERARCHYID. This is done via a recursive CTE query and a little path construction magic.

In the anchor query portion of the CTE, we are retrieving a root node using HIERARCHYID::GetRoot(). This establishes the root of the hierarchy and applies only to the root node (ManagerID IS NULL). In the recursive query portion, that same column is creating a new path based on the existing path of the parent, EHTemp.EmployeeNode.ToString(), and a generated relative path based on the partitioned row number results, CAST(EH.Position AS VARCHAR(30)) + '/'. The two pieces are concatenated and then cast as a HIERARCHYID data type, as shown here:

```
CAST(EHTemp.EmployeeNode.ToString()
    + CAST(EH.Position AS VARCHAR(30)) + '/' AS HIERARCHYID)
```

The following table shows how the path of the parent and the partitioned row number relate to the new path of each of the nodes in the hierarchy.

EmpID	MgrID	LastName	RowNum	Parent Path	Relative Path	New Path
109	NULL	Sánchez	1	NULL	N/A	N/A
6	109	Bradley	1	/	1/	/1/
2	6	Brown	1	/1/	1/	/1/1/
46	6	Harnpadoungsataya	2	/1/	2/	/1/2/
106	6	Gibson	3	/1/	3/	/1/3/
119	6	Williams	4	/1/	4/	/1/4/
203	6	Eminhizer	5	/1/	5/	/1/5/
269	6	Benshoof	6	/1/	6/	/1/6/
271	6	Wood	7	/1/	7/	/1/7/
272	6	Dempsey	8	/1/	8/	/1/8/
12	109	Duffy	2	/	2/	/2/
3	12	Tamburello	1	/2/	1/	/2/1/
4	3	Walters	1	/2/1/	1/	/2/1/1/
9	3	Erickson	2	/2/1/	2/	/2/1/2/
11	3	Goldberg	3	/2/1/	3/	/2/1/3/
158	3	Miller	4	/2/1/	4/	/2/1/4/
79	158	Margheim	1	/2/1/4/	1/	/2/1/4/1/
114	158	Matthew	2	/2/1/4/	2/	/2/1/4/2/
217	158	Raheem	3	/2/1/4/	3/	/2/1/4/3/
263	3	Cracium	5	/2/1/	5/	/2/1/5/
5	263	D'Hers	1	/2/1/5/	1/	/2/1/5/1/
265	263	Galvin	2	/2/1/5/	2/	/2/1/5/2/
267	3	Sullivan	6	/2/1/	6/	/2/1/6/
270	3	Salavaria	7	/2/1/	7/	/2/1/7/

When the newly created path is cast to HIERARCHYID, the *Parse* method is invoked behind the scenes, which takes a path and converts it to a HIERARCHYID data type.

Converting *XML* to a Hierarchy

Because I mentioned this earlier in the chapter, I feel compelled to include a code sample of converting an *XML* structure into a relational structure using HIERARCHYID.

```
DECLARE @x XML =
‘<A id=”1”>
    <B id=”2”>
        <C id=”3”/>
        <D id=”4”/>
    </B>
    <E id=”5”/>
    <F id=”6”/>
</A>’;

WITH Folders AS
(
SELECT
    t.c.value(‘@id’, ‘int’) AS ID
    , NULLIF(t.c.value(‘../@id’, ‘nvarchar(50)’), ‘’) AS ParentID
    , t.c.value(‘local-name(.)’, ‘nvarchar(50)’) AS Description
    , t.c.value(‘for $s in . return count(../*[. << $s]) + 1’, ‘int’) AS RowNum
FROM @x.nodes(‘//*’) AS t(c)
)
, FolderTree AS
(
    SELECT ID, ParentID, Description, RowNum,
        HIERARCHYID::GetRoot() AS FolderNode
    FROM Folders
    WHERE ParentID IS NULL

    UNION ALL

    SELECT F.ID, F.ParentID, F.Description, F.RowNum,
        CAST(FT.FolderNode.ToString() + CAST(F.RowNum AS varchar(50)) + ‘/’
            AS HIERARCHYID)
    FROM Folders AS F
        INNER JOIN FolderTree AS FT ON F.ParentID = FT.ID
)
SELECT ID, ParentID, Description, FolderNode, FolderNode.ToString()
FROM FolderTree
ORDER BY FolderNode;
```

You may be thinking to yourself, “Hey, this looks familiar,” and if you are, you are correct. Instead of using the ROW_NUMBER function in T-SQL, a little XQuery trick is used to achieve the same result, partitioned row numbers. Once that is included with an ID and ParentID, we

can use the same methodology as before to generate HIERARCHYID values for all rows in a single statement using a CTE and some basic path manipulation of the HIERARCHYID.

FILESTREAM

There are two camps when it comes to how to handle BLOB data. The first group stores the BLOB data in the database as *VARBINARY(MAX)*. This allows for complete control from SQL Server, more secure data, and better data integrity, and the data is included with SQL Server backups and restores.

The other group stores the BLOB data in the file system and stores the location of the file in the database. This methodology allows for easier and direct access to the data, which reduces the workload of SQL Server. The files can be stored in a different location on the network, reducing the network load of the SQL Server.

The new FILESTREAM feature in SQL Server 2008 works something like a hybrid of these two methodologies. It enables SQL Server applications to store the unstructured (BLOB) data directly on the file system (outside the database file) leveraging the rich streaming application programming interfaces (APIs) and high streaming performance of the file system. Compatibility with T-SQL programming is maintained. SQL FILESTREAM maintains transactional consistency between the unstructured data and the corresponding structured data without having to create custom cleanup logic at the application level. FILESTREAM also enables deep integration with T-SQL and manageability features—ACID properties (atomicity, consistency, isolation, and durability), triggers, full-text search, backup and restore, security, database consistency checks (DBCCs), and replication. Unstructured data can be accessed and manipulated using a transacted dual programming model: T-SQL and the Win32 streaming API set.

Before we delve into the finer points of FILESTREAM, however, we need to get it configured for use.

Configuring FILESTREAM

To use the FILESTREAM ability, it first needs to be configured. Configuration comes in two layers: one for the Windows layer and the other for the SQL Server instance layer. Configuring the Windows layer is done either during setup or, if post-setup, using SQL Server Configuration Manager. From SQL Server Configuration Manager, select SQL Server Services on the left, then right-click the SQL Server instance service, and choose Properties. Then select the FILESTREAM tab, as shown in Figure 3-4.

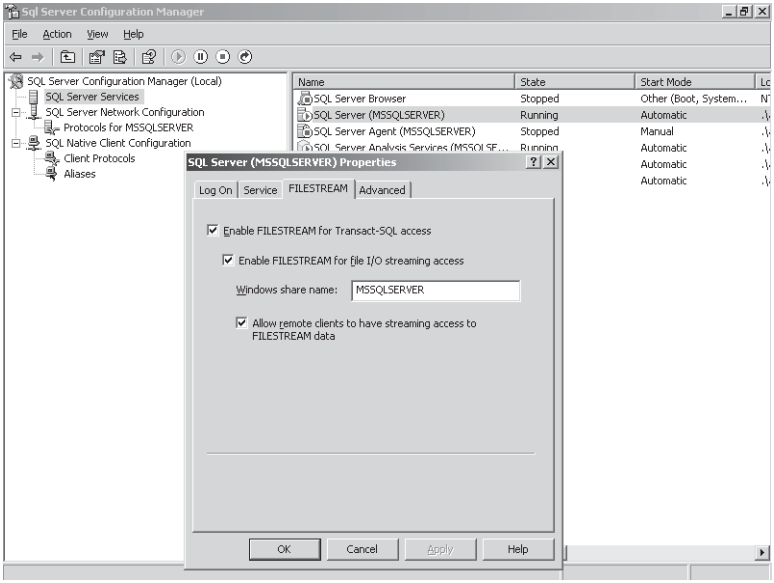


FIGURE 3-4 FILESTREAM configuration using SQL Server Configuration Manager

Note Configuring the Windows layer requires administrator privilege on the server.

The Windows layer settings allow you to specify which type of access—T-SQL or Win32 (file input/output, or I/O) streaming—and allow you to specify the share name used and if remote access to the file I/O stream is allowed. You should note that to enable file I/O stream access, you must first enable T-SQL access.

If you are unsure of the share name or level information, you can easily find out by executing this query:

```
SELECT SERVERPROPERTY ('FilestreamShareName') AS ShareName,
    SERVERPROPERTY ('FilestreamConfiguredLevel') AS ConfiguredLevel,
    SERVERPROPERTY ('FilestreamEffectiveLevel') AS EffectiveLevel;
```

Note You cannot change the share name unless you first disable FILESTREAM. Once disabled, you can re-enable with a different share name.

The share name you choose is indeed a Windows server level share and can be seen via the Computer Management administrator application, as shown in Figure 3-5.

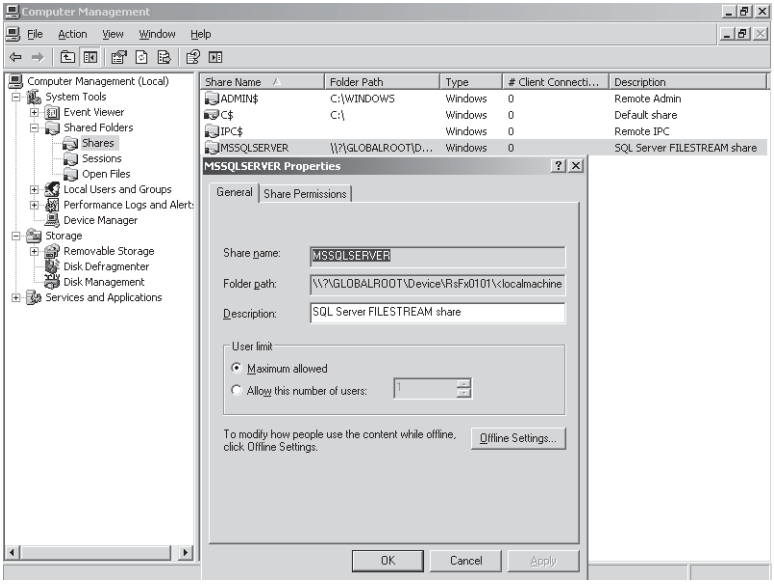


FIGURE 3-5 Computer Management Shares and Properties for the MSSQLSERVER share

The level of network access will affect the security on the share. If network access is enabled, then the share permissions are set for the special group Authenticated Users. If network access is disabled, however, then the share still exists, but the Everyone group is explicitly denied access to the share. And if file I/O stream access is disabled, the share is removed from the server altogether.

Once the Windows layer has been configured, you will still need to configure the SQL Server instance layer, which is done using the system stored procedure sp_configure. The following statement enables FILESTREAM, with T-SQL and Win32 streaming access.

```
EXEC sp_configure 'filestream access level', 2;
```

The configuration value determines both the state (enabled or disabled) as well as the access level, as shown in this table.

Configuration Value	Description
0	Disabled. This is the default value.
1	Enabled only for Transact-SQL access.
2	Enabled for Transact-SQL and Win32 streaming access.

This two-layered approach to configuration is done for security purposes. It requires that the server administrator and the database administrator (DBA) both enable access. Also, because

the DBA should not control which shares are created on a server, that responsibility is left to the server administrator. If the two settings differ, the least secure one is used. For example, if using SQL Server Configuration Manager, file I/O stream access is disabled at the Windows layer, but using sp_configure, it is enabled on the SQL Server instance layer, you will not be able to use file I/O to access the unstructured data via Win32 APIs.

Using FILESTREAM

Once FILESTREAM is enabled in both layers, you can begin using it. FILESTREAM is an extension to a *VARBINARY(MAX)* column definition, that allows the column to store the BLOB data in the file system.

Defining the Database

To use the feature, the database must have a FILESTREAM FILEGROUP defined. The following script creates a database named *FileStreamDatabase* that has the additional FILEGROUP that CONTAINS FILESTREAM.

```
USE master;
GO
IF EXISTS (
  SELECT * FROM sys.databases
  WHERE name = N'FileStreamDatabase'
)
  DROP DATABASE FileStreamDatabase
GO
CREATE DATABASE FileStreamDatabase ON PRIMARY
  ( NAME = FileStreamDatabase_data,
    FILENAME = N'C:\FileStreamDatabase\FileStreamDatabase_data.mdf',
    SIZE = 10MB,
    MAXSIZE = 50MB,
    FILEGROWTH = 15%),
FILEGROUP FileStreamDatabase_Filestream_FileGroup CONTAINS FILESTREAM
  ( NAME = FileStreamDatabase_Resumes,
    FILENAME = N'C:\FileStreamDatabase\Photos')
LOG ON
  ( NAME = 'FileStreamDatabase_log',
    FILENAME = N'C:\FileStreamDatabase\FileStreamDatabaseB_log.ldf',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB);
GO
```

You will notice that the FILENAME parameter for the FILESTREAM FILEGROUP is a path and not a specific file. The last folder in the path must not exist when creating the database, but the path up to (but not including) the last folder must exist. In this example above, the path

C:\FileStreamDatabase\ must exist, but C:\FileStreamDatabase\Photos\ must not exist. This folder is known as a *data container*.

Important It is important to ensure that only the SQL Server instance service account be allowed to access a data container (folder). You should use access control lists (ACLs) to protect the data container such that the only account with access is the service account, which is granted full rights to the data container.

Defining the Table

Once the database is created with a FILESTREAM FILEGROUP, you can define tables with FILESTREAM columns, as shown here:

```
CREATE TABLE dbo.Product
(
  ProductID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
  ProductName NVARCHAR(50) NOT NULL,
  ListPrice MONEY NOT NULL,
  ProductImage VARBINARY(MAX) FILESTREAM
);
```

I'm going to populate this table with some data from the *AdventureWorks* sample database:

```
INSERT INTO Product
SELECT NEWID(), P.Name, P.ListPrice, PP.ThumbNailPhoto
FROM AdventureWorks.Production.Product AS P
    INNER JOIN AdventureWorks.Production.ProductProductPhoto AS PPP
        ON P.ProductID = PPP.ProductID
    INNER JOIN AdventureWorks.Production.ProductPhoto AS PP
        ON PPP.ProductPhotoID = PP.ProductPhotoID
WHERE PP.ThumbnailPhotoFileName != 'no_image_available_small.gif'
```

You should also note that the table requires a unique UNIQUEIDENTIFIER column that is defined with ROWGUIDCOL. If the table does not have such a column and attempts to define a FILESTREAM, the CREATE TABLE statement will fail.

Note For tables that are not partitioned, all FILESTREAM columns are stored in a single FILESTREAM FILEGROUP. For tables that are partitioned, you use the FILESTREAM_ON option of the CREATE TABLE statement, and you must supply a partition scheme that uses the same partition function and partition columns as the partitions scheme for the table.

Also, although only a single FILESTREAM FILEGROUP can be used for non-partitioned tables, you can have multiple columns that specify the FILESTREAM option, as shown here:

```
CREATE TABLE dbo.ProductImage
(
  ProductID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
  SmallProductImage VARBINARY(MAX) FILESTREAM,
  LargeProductImage VARBINARY(MAX) FILESTREAM
);
```

Updating the Data

You can always update the data in the *VARBINARY(MAX)* column by using T-SQL. The INSERT statement in the previous section shows one example of such an action. As far as T-SQL is concerned, this is a *VARBINARY(MAX)* column and is treated as such. Behind the scenes, SQL Server is updating the data in a file in the file system instead of in a SQL Server data file.

When directly accessing the file, however, you still need to work with T-SQL. The process goes as follows:

1. Start a transaction.
    a. Get the file's path using the *PathName* method of the FILESTREAM column.
    b. Get the transaction context token using the GET_FILESTREAM_TRANSACTION_CONTEXT function.
2. Return the path and transaction token to the client.
3. Call the *OpenSqlFilestream* API method to obtain a Win32 handle using the path and transaction token.
    a. Access the file using the Win32 API (ReadFile, WriteFile, and so on).
4. Close the Win32 handle using the *CloseHandle* API method.
5. Commit the transaction (of course, you can also rollback).

The following pseudo-script demonstrates how Win32 file stream access is done:

```
--1: Begin a transaction.
BEGIN TRANSACTION;

--2: Get the path of a FILESTREAM file.
DECLARE @PathName nvarchar(max) =
    (
        SELECT TOP 1 ProductImage.PathName()
        FROM Product
        WHERE ProductID = '50d9dcfd-dd3c-4a48-ac6e-4e8f86986be9'
    );

--3: Get the transaction context.
```

```
DECLARE @TransactionToken varbinary(max);
SET @TransactionToken =  GET_FILESTREAM_TRANSACTION_CONTEXT ();

--At this point, you would return these values to the client
SELECT @PathName, @TransactionToken ;

/*
4: Call the OpenSqlFilestream API to obtain a Win32 handle
by passing in the path and transaction token from above. Next
using the Win32 hanlde you got from OpenSqlFilestream,
you can call the various Win32 API functions such as ReadFile(),
WriteFile(), TransitFile(), and so on. Of course, be sure to
call CloseHandle() when you are done working with the file.
*/

--5: You must then COMMIT or ROLLBACK the transaction.
COMMIT TRANSACTION;
```

For the client application to work with the file, you must return back from T-SQL two things: a logical path to the file and a transaction token. Although the path looks like a valid path, it is not usable except by the OpenSqlFilestream API. The transaction scope can be managed from the client and is required to return a transaction token. If you call the GET_FILESTREAM_TRANSACTION_CONTEXT function when not in a transaction, it returns NULL.

Spatial Data Types

I was recently involved in a project involving law enforcement (this is not a euphemism for being arrested), and, as part of the data architecture, we realized that it would be great if we could identify certain types of information on a map of the area in question—items such as locations of booted vehicles, sex offenders, or any number of other items.

Perhaps I'm working on a real estate system and want to be able to show in what school district or congressional district a house is located. Or maybe I want to be able to find retail stores in relation to a house, or maybe the nearest airports.

And so, may I introduce to you the new spatial data types of SQL Server 2008. The *GEOGRAPHY* data type is used to represent geodetic spatial data, items on the oblate spheroid, or slightly flattened sphere that is our planet Earth. The *GEOMETRY* data type represents planar spatial data, or items on flat surfaces. Both of these are implemented as system CLR types, so, like HIERARCHYID, they do not require you to enable CLR in order to use it.

Note This topic could fill a sizable book on its own, so bear in mind that we are only scratching the surface of spatial data in this introductory-level book.

Types of Spatial Data

As mentioned previously, there are two data types in SQL Server 2008 that can represent spatial data, *GEOMETRY* and *GEOGRAPHY*. There are a total of 11 different spatial data objects, although only 7 of them are instantiable: *Point*, *LineString*, *Polygon*, *GeometryCollection*, *MultiPoint*, *MultiLine*, and *MultiPolygon*. Figure 3-6 shows the entire hierarchy of all 11 (the instantiable ones are shaded dark gray).

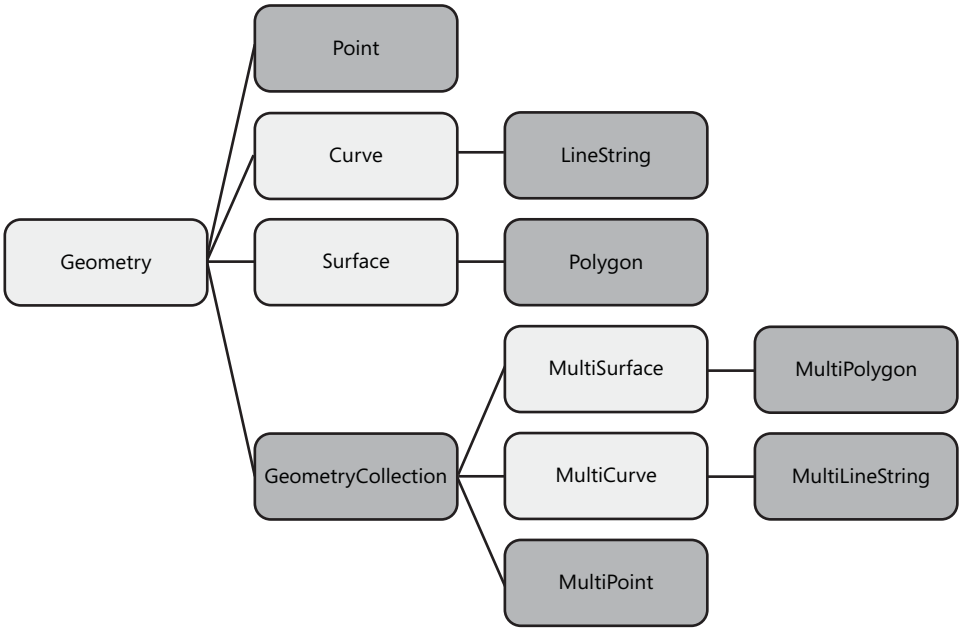


FIGURE 3-6 Spatial data type hierarchy

Working with the Spatial Data Types

To best show how these various instant types work, I am going to use the following shapes (numbered 1 through 8):

1. *Polygon*: A square with a small hole in the middle.
2. *Polygon*: A square located within polygon 1.
3. *Polygon*: A square located in the hole of polygon 1.
4. *Polygon*: A pentagon that, although is within the outer boundaries of polygon 1, covers parts of polygon 1 and the hole within polygon 1.
5. *Polygon*: A square that shares a border with polygon 1.
6. *LineString*: A line that touches polygon 1.

7. *MultiLineString*: Two lines, one of which crosses polygons 1 and 5 and the other which touches polygon 5.
8. *MultiPolygon*: Four squares, three of which are wholly contained within polygon 1 and a fourth which lies outside of polygon 1.

Figure 3-7 shows how these spatial instances relate to each other.

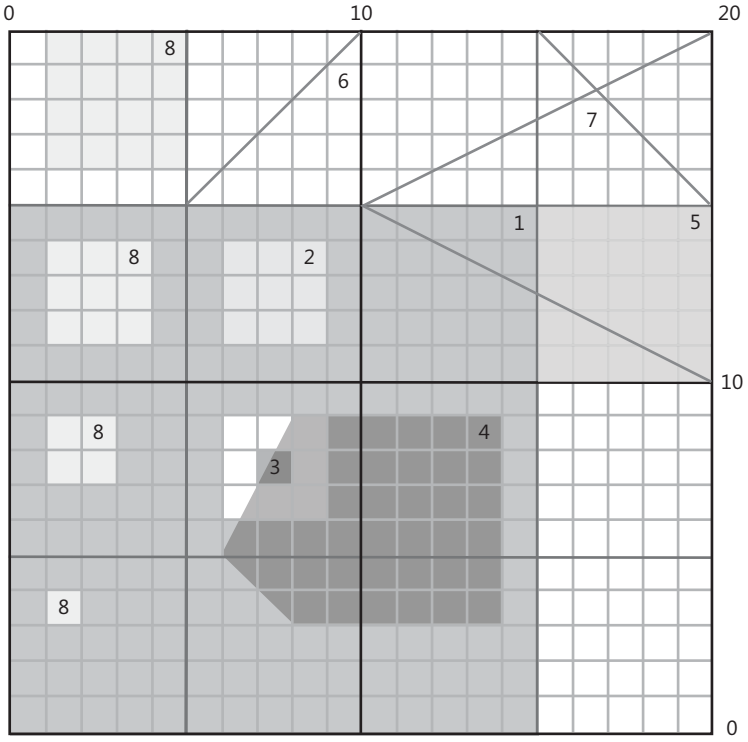


FIGURE 3-7 A collection of various spatial instance types

The following script creates and loads the table that holds these eight *GEOMETRY* object instances.

```
USE TempDB
GO

IF EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Shape]') AND
type in (N'U'))
    DROP TABLE [dbo].[Shape]
GO

CREATE TABLE [dbo].[Shape]
(
```

```
    ShapeID INT IDENTITY(1,1) NOT NULL,
    Shape GEOMETRY NOT NULL
)
GO

INSERT INTO Shape (Shape)
VALUES
  (GEOMETRY::STGeomFromText
    ('POLYGON((0 0, 15 0, 15 15, 0 15, 0 0),(6 6, 6 9, 9 9, 9 6, 6 6))', 0)),
  (GEOMETRY::STGeomFromText('POLYGON((6 11, 6 14, 9 14, 9 11, 6 11))', 0)),
  (GEOMETRY::STGeomFromText('POLYGON((7 7, 7 8, 8 8, 8 7, 7 7))', 0)),
  (GEOMETRY::STGeomFromText('POLYGON((8 3, 6 5, 8 9, 14 9, 14 3, 8 3))', 0)),
  (GEOMETRY::STGeomFromText('POLYGON((15 10, 15 15, 20 15, 20 10, 15 10))', 0)),
  (GEOMETRY::STGeomFromText('LINESTRING(5 15, 10 20)', 0)),
  (GEOMETRY::STGeomFromText
    ('MULTILINESTRING((20 10, 10 15, 20 20),(20 15, 15 20))', 0)),
  (GEOMETRY::STGeomFromText
      ('MULTIPOLYGON(
          ((1 16, 1 20, 5 20, 5 16, 1 16)),
          ((1 11, 1 14, 4 14, 4 11, 1 11)),
          ((1 7, 1 9, 3 9, 3 7, 1 7)),
          ((1 3, 1 4, 2 4, 2 3, 1 3)) )'
      , 0));
```

Now we just need to know what we want to know about these shapes. Perhaps you want to know the area, maximum dimension, and the geometry type of each of these shapes. A simple query like the following could tell you:

```
SELECT
    ShapeID,
    Shape.STArea() AS Area,
    Shape.STDimension() AS Dimension,
    Shape.STGeometryType() AS GeometryType
FROM Shape;
```

Executing the preceding query gives you the following results:

ShapeID	Area	Dimension	GeometryType
1	216	2	*Polygon*
2	9	2	*Polygon*
3	1	2	*Polygon*
4	42	2	*Polygon*
5	25	2	*Polygon*
6	0	1	*LineString*
7	0	1	*MultiLineString*
8	30	2	*MultiPolygon*

These methods (along with several others) are used to report back information about a shape instance. However, what if you want to compare two different shape instances? Many of the methods available for the *GEOMETRY* data type allow comparisons such as:

```
SELECT
    S1.ShapeID AS Shape1
    ,S2.ShapeID AS Shape2
    ,S1.Shape.STCrosses(S2.Shape) AS STCrosses
    ,S1.Shape.STIntersects(S2.Shape) AS STIntersects
    ,S1.Shape.STTouches(S2.Shape) AS STTouches
    ,S1.Shape.STWithin(S2.Shape) AS STWithin
    ,S1.Shape.STContains(S2.Shape) AS STContains
    ,S1.Shape.STEquals(S2.Shape) AS STEquals
    ,S1.Shape.STDifference(S2.Shape).STAsText() AS STDifference
    ,S1.Shape.STIntersection(S2.Shape).STAsText() AS STIntersection
    ,S1.Shape.STUnion(S2.Shape).STAsText() AS STUnion
    ,S1.Shape.STDistance(S2.Shape) AS STDistance
FROM Shape AS S1
    CROSS JOIN Shape AS S2
WHERE S1.ShapeID > S2.ShapeID
ORDER BY S1.ShapeID, S2.ShapeID
```

This query compares all the shape instances with each and returns the results from the following methods:

Method Name	Description
STIntersects	Returns *True* if the two shape instances intersect, even if by a single point. When *True*, the corresponding *STIntersection* method will always return some shape instance. When *False*, *STIntersection* returns an empty *GeometryCollection*.
STWithin	Returns *True* if the shape instance that is calling the method falls entirely within the shape instance that is passed in as the method parameter. Unlike most methods, the order of the shape instances is important for this method.
STContains	The reverse of the *STWithin* method; returns *True* if the shape instance that is calling the method entirely contains the shape instance that is passed in as the method parameter. The only time both *STWithin* and *STContains* can both return *True* for the same set of shape instances is when the two shape instances are equal.
STEquals	Returns *True* if the two shape instances are identical.
STDifference	Returns the difference of the two shape instances as a shape.
STIntersection	Returns the intersection of the two shape instances as a shape.
STUnion	Returns the union of the two shape instances as a shape.
STDistance	Returns the closest distance between the two shape instances.

Before I move on, I'd like to show some of the results from *STDifference*, *STIntersection*, and *STUnion*. For example, shape 1 and shape 4 are defined as follows:

- **Shape 1** POLYGON ((0 0, 15 0, 15 15, 0 15, 0 0), (6 6, 6 9, 9 9, 9 6, 6 6))
- **Shape 2** POLYGON ((8 3, 6 5, 8 9, 14 9, 14 3, 8 3))

And here are the methods and the return values:

- ***STDifference*** POLYGON ((0 0, 15 0, 15 15, 0 15, 0 0), (8 3, 6 5, 6.5 6, 6 6, 6 9, 8 9, 9 9, 14 9, 14 3, 8 3))
- ***STIntersection*** GEOMETRYCOLLECTION (LINESTRING (9 9, 8 9), POLYGON ((8 3, 14 3, 14 9, 9 9, 9 6, 6.5 6, 6 5, 8 3)))
- ***STUnion*** POLYGON ((0 0, 15 0, 15 15, 0 15, 0 0), (6 6, 6 9, 8 9, 6.5 6, 6 6))

As shown above, the *STDifference* and *STUnion* methods both return a *Polygon*, but the *STIntersection* method returns a *GeometryCollection*. Figure 3-8 shows what the resulting shape instance intersection looks like.

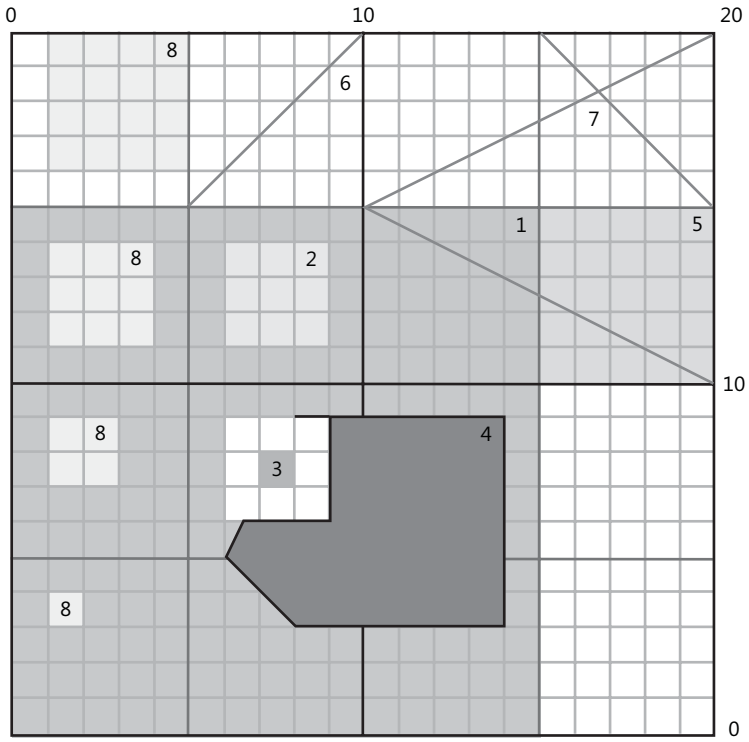


FIGURE 3-8 The resulting *GEOMETRY* from the *STIntersection* method

Notice that there is a small line protruding from the upper left corner of the polygon labeled 4. Because a polygon cannot contain such a line, the results are a *LineString* from point (8, 9) to point (9, 9), and the remainder is the *Polygon*.

Spatial Indexing

Now that you have these two new spatial data types, how can you use them efficiently? Like the *XML* data type, spatial types have a special indexing ability that is focused on optimizing the querying abilities, as well as the storage of said index data.

Spatial indexes use a decomposition methodology that partitions the area being indexed into a grid. A second layer (or level) contains grid cells from the first layer, each of which is broken down into another grid. This process continues for a total of four layers, each containing a more detailed breakdown of the previous layer. By default, each layer uses an 8x8 grid breakdown as its basis, which means a default spatial index would be decomposed as follows:

- Level 1 would be an 8x8 grid, or 64 cells.
- Level 2 would have an 8x8 grid for each of the 64 cells in level 1, or 4,096 cells.
- Level 3 would have an 8x8 grid for each of the 4,096 cells in level 2, or 262,144 cells.
- Level 4 would have an 8x8 grid for each of the 262,144 cells in level 3, or 16,777,216 cells.

When creating the index, you can override the density of the grid at each level. There are three levels of density available to use:

- **LOW** Uses a 4x4 grid for a total of 16 cells
- **MEDIUM** Uses an 8x8 grid, or 64 cells
- **HIGH** Uses a 16x16 grid, or 256 cells

Once the index decomposes the space into the four levels, it then begins to tessellate each spatial object on a row-by-row basis. Tessellation is the process that fits the object into the grid levels, starting at level 1, and working down until the Cells-Per-Object rule states that it should stop processing the object. You can use the spatial index's CELLS_PER_OBJECT option to any integer value from 1 to 8,192.

The default for the CELLS_PER_OBJECT option is 16; this default value generally provides a good balance between the index precision and the size of the index. As you increase this number, the precision increases but so does the size of the index, and a large index can adversely affect performance.

More Info The tessellation process for both the GEOMETRY and GEOGRAPHY data types is fascinating, but alas, it is also beyond the scope of this book. For further reading on how SQL Server breaks down a plane or the earth into increasingly denser grids, see the "Spatial Indexing Overview" topic in SQL Server 2008 Books Online.

Spatial Index Rules and Restrictions

Like all features, the ability of spatial indexes is not without limit.

- The table in which you create the spatial index must have a clustered primary key with 15 or fewer key columns.
- You can have multiple spatial indexes (up to 249) per table.
- Spatial indexes are not usable with indexed views.
- When using a GEOMETRY_GRID tessellation, you can additionally supply a bounding box for the index. These options are not available for GEOGRAPHY_GRID because it uses the whole globe as its bounding box.

Creating a Spatial Index

You can create a spatial index using SQL Server Management Studio (SSMS) or T-SQL. For example, using the following T-SQL code, I can create a spatial index on a *GEOGRAPHY* data type column that uses LOW density for levels 1 and 4, uses MEDIUM density for levels 2 and 3, and uses at most 16 cells to define any object in the index:

```
CREATE SPATIAL INDEX [IXS_KeyGeographicLocation_KeyGeographicLocation]
ON [dbo].[KeyGeographicLocation]
(
  [KeyGeographicLocation]
)
USING  GEOGRAPHY_GRID
WITH
(
  GRIDS =(LEVEL_1 = LOW,LEVEL_2 = MEDIUM,LEVEL_3 = MEDIUM,LEVEL_4 = LOW),
  CELLS_PER_OBJECT = 16
);
```

Or, in SSMS Object Explorer, I can right-click on the Indexes node under the KeyGeographicLocation table and choose New Index. The New Index dialog box opens, as shown in Figure 3-9.

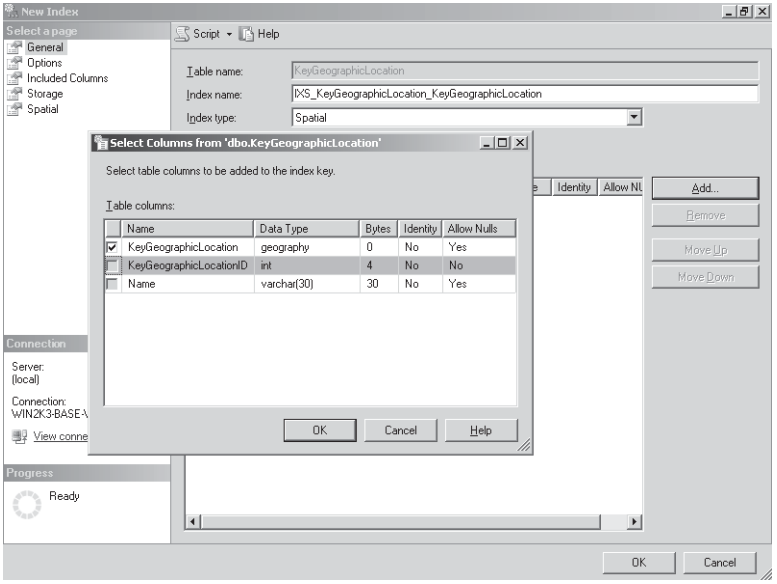


FIGURE 3-9 New Index and Select Columns dialog boxes

Figure 3-9 also shows that when choosing a spatial index type, the column selection is limited to only spatial data type columns; columns of other data types are disabled.

Figure 3-10 shows how to set the special spatial index options, such as grid densities and cells per object.

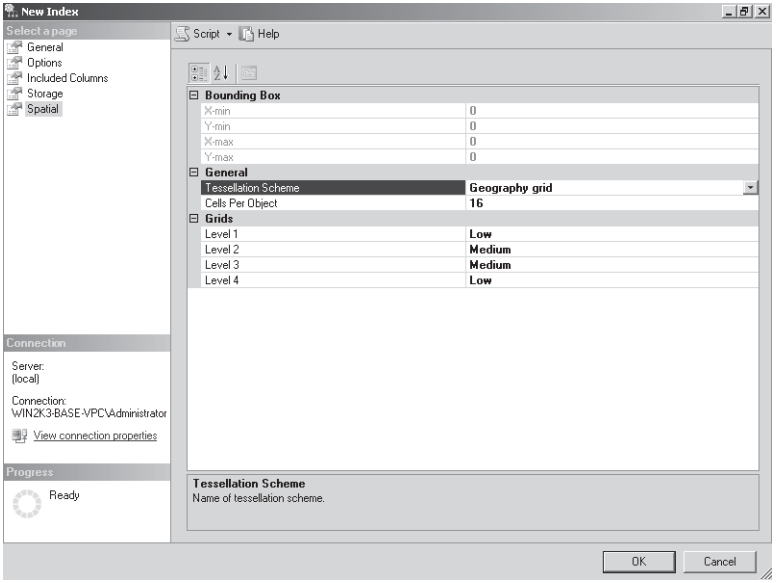


FIGURE 3-10 New index spatial options

Keep in mind that because I chose a tessellation for *GEOGRAPHY*, the bounding box features are disabled. If I had been working with a *GEOMETRY* data type instead, I would choose Geometry Grid as the tessellation scheme, and the bounding box options would be enabled.

Spatial in the World

All of the examples shown so far in this section have been for the *GEOMETRY* data type, so I thought a good way to round out this chapter is to see some practical use of the *GEOGRAPHY* data type. I have loaded several sets of *GEOGRAPHY* data into a database called *NewYorkCity*, which contains the spatial data and related metadata for the roads and an abridged set of key landmarks and locations in Manhattan. In this particular set of data, roads, landmarks, and locations are represented as one or more shapes, so, for example, Columbia University is actually spread over two rows of data, as shown by this query.

```
SELECT LandmarkID, Landmark.ToString() AS LandmarkText, Name
FROM dbo.LANDMARK
WHERE NAME = 'Columbia University'
```

This SELECT statement returns the following data.

LandmarkID	LandmarkText	Name
36	POLYGON ((-73.960162 40.808155, -73.961079 40.806824, -73.963956 40.808001999999995, -73.962054999999992 40.810631, -73.959223 40.809402999999996, -73.95966 40.808794999999996, -73.960162 40.808155))	Columbia University
352	POLYGON ((-73.961079 40.806824, -73.961603 40.806166999999995, -73.96204 40.805558999999995, -73.964894 40.806754, -73.964392 40.807410999999995, -73.963956 40.808001999999995, -73.961079 40.806824))	Columbia University

So what can we do with this data? First, let's see how big the university actually is.

```
SELECT SUM(Landmark.STArea()) AS LandmarkArea
FROM dbo.LANDMARK
WHERE NAME = 'Columbia University'
```

This query returns a value of 49441.0036718398 (square meters). Perhaps we want to see what other landmarks are within a half mile of the university.

```
;WITH Columbia AS
(    SELECT *
    FROM dbo.LANDMARK
    WHERE NAME = 'Columbia University'
)
SELECT L.Name, MIN(L.Landmark.STDistance(C.Landmark)) AS Distance
FROM dbo.Landmark AS L
    INNER JOIN Columbia AS C
        ON L.Landmark.STDistance(C.Landmark) <= 804 -- 804 meters = ~1/2 mile
        AND L.Name != C.Name
GROUP BY L.Name
ORDER BY MIN(L.Landmark.STDistance(C.Landmark))
```

This returns the following results.

Name	Distance
Barnard College	0
Morningside Park	84.6680452663973
Riverside Park	98.5865762586118
Hudson River	216.701882646377
Central Park	289.377374567307
The Reservoir	551.437153705882
Harlem Mere	651.499219450634
The Mt Sinai Medical Center	785.846014446101

Looks like I'll be heading over to Morningside Park today.

Now, a quick quiz: In the following two statements, will the resulting *GEOGRAPHY* shapes be the same or different? How do the *Parse* and *STGeomFromText* static *GEOGRAPHY* methods differ?

```
DECLARE @Columbia1 GEOGRAPHY =
GEOGRAPHY::STGeomFromText('POLYGON ((-73.960162 40.808155,
-73.961079 40.806824, -73.963956 40.808001999999995,
-73.962054999999992 40.810631, -73.959223 40.809402999999996,
-73.95966 40.808794999999996, -73.960162 40.808155))', 4236)

DECLARE @Columbia1 GEOGRAPHY =
GEOGRAPHY::Parse('POLYGON ((-73.960162 40.808155,
-73.961079 40.806824, -73.963956 40.808001999999995,
-73.962054999999992 40.810631, -73.959223 40.809402999999996,
-73.95966 40.808794999999996, -73.960162 40.808155))')
```

For the first question, if you said the shapes are the same, then you are correct. Unlike *GEOMETRY* shapes, *GEOGRAPHY* shapes require a valid spatial reference identifier (SRID). *Parse* and *STGeomFromText* differ only in that *STGeomFromText* requires an SRID, whereas *Parse* assumes an SRID of 4326 (the default spatial reference system known by the name World Geodetic System 1984), which happens to use the meter as its base unit of measurement.

Because SRID determines the base unit of measurement, type of spatial model, and the ellipsoid used for the flat-earth or round-earth mapping, you can only compare shapes with the same SRID. To determine the SRID of a shape, use the *STSrid* property of your shape, for example:

```
SELECT @Columbia1.STSrid
```

Also note that if you used the wrong SRID when loading your data, you can change the SRID of your data by assigning the *STSrid* property the correct value, as shown here:

```
UPDATE Road SET Road.STSrid = 4267
```

As much as this section may have shown you about the new spatial data types, it has only scratched the surface of the technology and possible uses.

XML Data Type

SQL Server 2005 introduced *XML* as a first-class data type. SQL Server 2008 extends the *XML* data type abilities in both schema validation support and XQuery capabilities.

XML Schema Validation Enhancements

On the schema validation front, there are three new features that need to be discussed: lax validation, lists and unions, and date and time data.

Lax Validation support

I have been anxiously awaiting lax validation support since the *XML* data type was introduced in SQL Server 2005. Lax validation gives you the ability to strongly type those parts of your *XML* data that need to be strongly typed, while allowing you to have portions that are weakly typed. Usually you will have a majority of your schema that is strongly typed and only a small part that allows for any type.

Note The examples and demonstration in this section focus on lax validation with *XML* elements but could also be implemented for attributes.

Let's start by defining the following schema collection using skip validation:

```
CREATE XML SCHEMA COLLECTION xsProductOrder
AS
'<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Products">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Product">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" />
              <xs:any namespace="##any" processContents="skip"
                      minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="ProductID" type="xs:unsignedShort" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>'
```

This schema collection states that there must be a *Name* element under the *Product* element, which can optionally be followed by any number of elements, regardless of namespace, none of which will be validated (because of the skip validation). The following *XML* would be valid using this schema.

```
DECLARE @X XML (xsProductOrder) =
'<Products>
  <Product ProductID="897">
    <Name>LL Touring Frame - Blue, 58</Name>
    <Order OrderID="51823" />
    <Order OrderID="51875" />
  </Product>
  <Product ProductID="942" >
    <Name>ML Mountain Frame-W - Silver, 38</Name>
    <Order OrderID="51120" />
    <Order OrderID="51711" />
    <Order OrderID="51758" />
    <Order OrderID="51799" />
    <Order OrderID="51856" />
  </Product>
</Products>'
```

The namespace of the *Any* element can be a whitespace separated list of uniform resource identifiers (URIs). The following special values can also be used:

Namespace	Meaning
##any	*XML* from any namespace is valid (default). Cannot be combined with other namespaces.
##other	*XML* from a qualified namespace other than the target namespace of the type being defined. Cannot be combined with other namespaces.
##targetNamespace	*XML* from the target namespace. Can be included in the whitespace separated list.
##local	*XML* not declared in a namespace (unqualified). Can be included in the whitespace separated list.

To better understand these other namespace options, let's look at one more example, this time using lax validation:

```
CREATE XML SCHEMA COLLECTION xsProductOrder
AS
'<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://debetta.com/xml/products/"
           targetNamespace="http://debetta.com/xml/products/">
  <xs:element name="Products">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Product">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" />
              <xs:any namespace="##other" processContents="lax"
                      minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="ProductID" type="xs:unsignedShort" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>'
```

The elements of this schema are now defined to be part of the *http://debetta.com/xml/products/namespace*. The *Any* element, however, must be from a different namespace, so although this would be valid:

```
DECLARE @X XML (xsProductOrder) =
'<Products xmlns="http://debetta.com/xml/products/">
  <Product ProductID="897">
```

```
    <Name>LL Touring Frame - Blue, 58</Name>
  </Product>
  <Product ProductID="942" >
    <Name>ML Mountain Frame-W - Silver, 38</Name>
  </Product>
</Products>'
```

This *XML* assignment will not work:

```
DECLARE @X XML (xsProductOrder) =
'<Products xmlns="http://debetta.com/xml/products/">
  <Product ProductID="897">
    <Name>LL Touring Frame - Blue, 58</Name>
    <Order OrderID="51823" />
    <Order OrderID="51875" />
  </Product>
  <Product ProductID="942" >
    <Name>ML Mountain Frame-W - Silver, 38</Name>
    <Order OrderID="51120" />
    <Order OrderID="51711" />
    <Order OrderID="51758" />
    <Order OrderID="51799" />
    <Order OrderID="51856" />
  </Product>
</Products>'
```

To fix the problem, we need to define and use a second namespace for the *Order* element, because the *Any* element must be from some other namespace, as shown here:

```
DECLARE @X XML (xsProductOrder) =
'<Products xmlns:O="http://debetta.com/xml/orders/"
           xmlns="http://debetta.com/xml/products/">
  <Product ProductID="897">
    <Name>LL Touring Frame - Blue, 58</Name>
    <O:Order OrderID="51823" />
    <O:Order OrderID="51875" />
  </Product>
  <Product ProductID="942" >
    <Name>ML Mountain Frame-W - Silver, 38</Name>
    <O:Order OrderID="51120" />
    <O:Order OrderID="51711" />
    <O:Order OrderID="51758" />
    <O:Order OrderID="51799" />
    <O:Order OrderID="51856" />
  </Product>
</Products>'
```

Finally, because the schema is doing lax validation on the *Any* element, if it finds the schema information, it will validate it; otherwise, it will not. If we had used the following schema instead, the *XML* assignment immediately above would fail because *Order* is now defined in another schema, so it can now be checked. And it requires an *OrderNum* attribute, not an *OrderID* attribute.

```
CREATE XML SCHEMA COLLECTION xsProductOrder
AS
'<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://debetta.com/xml/orders/"
           targetNamespace="http://debetta.com/xml/orders/">
  <xs:element name="Order">
    <xs:complexType>
      <xs:attribute name="OrderNum" type="xs:unsignedShort" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://debetta.com/xml/products/"
           targetNamespace="http://debetta.com/xml/products/">
  <xs:import namespace="http://debetta.com/xml/orders/" />
  <xs:element name="Products">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Product">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" />
              <xs:any namespace="##other" processContents="lax"
                      minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="ProductID" type="xs:unsignedShort"
                          use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>'
```

Union and List Type Improvements

In SQL Server 2005, you could define a list as a type and use it to limit the valid values for an element or attribute, but it could only be a single list of items. What if you wanted to instead have two different lists? Well, in SQL Server 2005, you would have to define each one separately, and this implementation was not very flexible. In SQL Server 2008, however, you can

define a union of lists. In this example, the element *ProductType* can either be *Inventory* and/or *DirectShip* or it can be *Electronic* and/or *Physical*.

```
CREATE XML SCHEMA COLLECTION xsProductOrder
AS
'<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://debetta.com/xml/orders/"
           targetNamespace="http://debetta.com/xml/orders/">
  <xs:element name="Order">
    <xs:complexType>
      <xs:attribute name="OrderID" type="xs:unsignedShort" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://debetta.com/xml/products/"
           targetNamespace="http://debetta.com/xml/products/">
  <xs:import namespace="http://debetta.com/xml/orders/" />
  <xs:element name="Products">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Product">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Name" type="xs:string" />
              <xs:element ref="ProductType" />
              <xs:any namespace="##other" processContents="lax"
                      minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="ProductID" type="xs:unsignedShort"
                          use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ProductType">
    <xs:simpleType>
      <xs:union>
        <xs:simpleType>
          <xs:list>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="Inventory"/>
                <xs:enumeration value="DirectShip"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:list>
        </xs:simpleType>
```

```
        <xs:simpleType>
          <xs:list>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="Electronic"/>
                <xs:enumeration value="Physical"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:list>
        </xs:simpleType>
      </xs:union>
    </xs:simpleType>
  </xs:element>
</xs:schema>'
```

This declaration is therefore valid:

```
DECLARE @X XML (xsProductOrder) =
'<Products xmlns:O="http://debetta.com/xml/orders/"
           xmlns="http://debetta.com/xml/products/">
<Product ProductID="897">
  <Name>Cycle Racer Simulation Software</Name>
  <ProductType>Physical Electronic</ProductType>
  <O:Order OrderID="51823" />
  <O:Order OrderID="51875" />
</Product>
<Product ProductID="942" >
  <Name>ML Mountain Frame-W - Silver, 38</Name>
  <ProductType>Inventory</ProductType>
  <O:Order OrderID="51120" />
  <O:Order OrderID="51711" />
  <O:Order OrderID="51758" />
  <O:Order OrderID="51799" />
  <O:Order OrderID="51856" />
</Product>
</Products>'
```

There are methods for achieving similar results in SQL Server 2005, but they are more cumbersome and require defining each list as a separate type.

DateTime, Date, and Time Validation

Although the *XML* schema specification states that time zone information is optional, SQL Server 2005 requires it. SQL Server 2008 removes this restriction to better comply with the specification. Additionally, SQL Server 2005 would convert the time zone to Coordinated Universal Time (UTC). The following example shows such a conversion from SQL Server 2005:

```
CREATE XML SCHEMA COLLECTION MySampleCollection AS '
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://debetta.com/"
    targetNamespace="http://debetta.com/">
  <xs:element name="OrderDate" type="xs:dateTime"/>
</xs:schema>'
GO

DECLARE @x XML(MySampleCollection)
SET @x =
  '<OrderDate xmlns="http://debetta.com/">1999-05-31T13:20:00-05:00</OrderDate>'
SELECT @x
```

This would return the following result:

```
<OrderDate xmlns="http://debetta.com/">1999-05-31T18:20:00Z</OrderDate>
```

SQL Server 2008, however, preserves time zone information, so no conversions occur. In SQL Server 2008, that same script would return the following:

```
<OrderDate xmlns="http://debetta.com/">1999-05-31T13:20:00-05:00</OrderDate>
```

SQL Server 2008 does add a restriction to date data in that the year must now range from 1 to 9999, whereas in SQL Server 2005, it could range from -9999 to 9999. However, time data now has 100 nanosecond precision, far greater than that available in SQL Server 2005.

Note When upgrading from SQL Server 2005 to 2008, date, time, and dateTime types can be affected. For more information, see the "Upgrading Typed *XML* from SQL Server 2005 to SQL Server 2008" section in the "Typed *XML* Compared to Untyped *XML*" topic in SQL Server 2008 Books Online.

XQuery

Three new features of note have been added to XQuery in SQL Server 2008: new functions, variable assignment, and enhanced relational data binding.

Changing Case

Two new functions have been added to the XQuery repertoire: fn:upper-case() and fn:lower-case(). As you may know, *XML* is case sensitive, so having these case-changing functions will certainly be of use.

Let Clause

FLWOR is an acronym for the XQuery clauses *for*, *let*, *where*, *order by*, and *return*. SQL Server 2005 supported all but the *let* clause; SQL Server 2008 adds support for *let*. The *let* clause is a simple variable [declaration and] assignment. This example uses *let* to calculate a count of the *Order* nodes:

```
DECLARE @x XML =
'<Product ProductID="897">
    <Name>LL Touring Frame - Blue, 58</Name>
    <Order OrderID="51823" />
    <Order OrderID="51875" />
</Product>
<Product ProductID="942" >
    <Name>ML Mountain Frame-W - Silver, 38</Name>
    <Order OrderID="51120" />
    <Order OrderID="51711" />
    <Order OrderID="51758" />
    <Order OrderID="51799" />
    <Order OrderID="51856" />
</Product>'

SELECT @x.query(
'<Products>
{
for $order in /Product
let $count :=count($order/Order)
order by $count
return
<Product>
{$order/Name}
<OrderCount>{$count}</OrderCount>
</Product>
}
</Products>')
```

In this example, the *let* clause is executed twice, once for each *Product*. You could also have implemented the following XQuery to produce the same results:

```
SELECT @x.query(
'<Products>
{
for $order in /Product
return
<Product>
{$order/Name}
<OrderCount>{count($order/Order)}</OrderCount>
</Product>
}
</Products>')
```

Although both of the queries return the same results, as the expression gets more complicated, or if the value needs to be used multiple times, it is advantageous to use a variable assigned via the *let* clause than to embed a more complicated expression or to repeat an expression.

Note The *let* clause cannot be used with constructed *XML* elements.

XML Data Manipulation Language

Yet another great new ability for *XML* in SQL Server 2008 is using the *sql:column* and *sql:variable* in modify method to be an *XML* construct. In SQL Server 2005, you could use these two functions in XQuery data manipulation language (DML), but they could only represent data and could not be used to insert one *XML* variable into another *XML* variable (a problem that I had to deal with on several occasions). To achieve this result, a little shred-and-recompose methodology was used to build the combined *XML*. Although this solution works, it requires that you first shred the *XML* data using the nodes, values, and query methods of the *XML* data type and then recompose the now relational form of the data back into *XML*. Here is an example of one such solution.

```
DECLARE @x1 XML =’
<items>
      <item type=”dynamic” id=”a”/>
      <item type=”static” id=”b”>
            <subitems>
                  <subitem id=”1” />
                  <subitem id=”2” />
                  <subitem id=”3” />
            </subitems>
      </item>
</items>’;

DECLARE @x2 XML = ‘
<subitems>
      <subitem id=”4” />
      <subitem id=”5” />
</subitems>’;

SELECT @x1 =
(SELECT
    items.itemXML.value(‘@type’, ‘NVARCHAR(255)’) AS [@type],
    items.itemXML.value(‘@id’, ‘NVARCHAR(5)’) AS [@id],
    items.itemXML.query(‘subitems’) AS [*],
    CASE items.itemXML.value(‘./@type’, ‘VARCHAR(255)’)
        WHEN ‘dynamic’ THEN @x2
        END AS [*]
FROM @x1.nodes(‘/items/item’) AS items(itemXML)
FOR XML PATH(‘item’), ROOT(‘items’), TYPE);
```

In SQL Server 2008, you have a much better path to take, for now you can simply update the *XML* data using the update method of the *XML* data type and add the *XML* in the second variable to the first variable. The biggest benefit here is that the *XML* data is updated in place, so when adding a small chunk of *XML* to a large document, much less work needs to be done. Thus, it will be faster than the shred-and-recompose method. The new solution is shown here:

```
DECLARE @x1 XML =’
<items>
      <item type=”dynamic” id=”a”/>
      <item type=”static” id=”b”>
            <subitems>
                  <subitem id=”1” />
                  <subitem id=”2” />
                  <subitem id=”3” />
            </subitems>
      </item>
</items>’;

DECLARE @x2 XML = ‘
<subitems>
      <subitem id=”4” />
      <subitem id=”5” />
</subitems>’;

SET @x1.modify
(
    ‘insert sql:variable(“@x2”) as first into (/items/item[@type=”dynamic”])[1]’
);
```

Now isn’t that so much nicer?

New Date and Time Data Types

SQL Server 2008 has extended the type system to include four new data and time data types: *DATE*, *TIME*, *DATETIME2*, and *DATETIMEOFFSET*. These new date and time data types offer higher precision, large year range, and time zone awareness and preservation and therefore more flexibility when defining your data domain. The following table gives a summary of the new and existing date and time data types:

Data Type	Primary Literal String Format	Range	Accuracy	Storage Size (bytes)
TIME [(0~7)]	hh:mm:ss [nnnnnnn]	00:00:00.0000000 through 23:59:59.9999999	100 nano-seconds	3 to 5
DATE	YYYY-MM-DD	0001-01-01 through 9999-12-31	1 day	3

Data Type	Primary Literal String Format	Range	Accuracy	Storage Size (bytes)
SMALLDATETIME	YYYY-MM-DD hh:mm:ss	1900-01-01 through 2079-06-06	1 minute	4
DATETIME	YYYY-MM-DD hh:mm:ss[.nnn]	1753-01-01 through 9999-12-31	0.00333 second	8
DATETIME2[(0~7)]	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nano-seconds	6 to 8
DATETIMEOFFSET[(0~7)] (Stores Time Zone Offset)	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nano-seconds	8 to 10

This table shows that the existing *DATETIME* and *SMALLDATETIME* data types haven't changed. The new date data types, however, now support a date range from 0001-01-01 to 9999-12-31. And the new time data types have precision up to 100 nanoseconds—that's 33,333 times more accuracy.

In addition, time data type can specify the precision, and, to support his feature, the three new data types that support time have an optional precision parameter. Take a look at the following code:

```
DECLARE
    @t0 TIME(0) = '12:34:56.1234567',
    @t1 TIME(1) = '12:34:56.1234567',
    @t2 TIME(2) = '12:34:56.1234567',
    @t3 TIME(3) = '12:34:56.1234567',
    @t4 TIME(4) = '12:34:56.1234567',
    @t5 TIME(5) = '12:34:56.1234567',
    @t6 TIME(6) = '12:34:56.1234567',
    @t7 TIME(7) = '12:34:56.1234567',
    @t TIME = '12:34:56.1234567'

SELECT
    @t0 AS T0,
    @t1 AS T1,
    @t2 AS T2,
    @t3 AS T3,
    @t4 AS T4,
    @t5 AS T5,
    @t6 AS T6,
    @t7 AS T7,
    @t AS T
```

Results of this query are shown in the following tables (results are split into two tables for formatting purposes):

T0	T1	T2	T3	T4
12:34:56	12:34:56.1000000	12:34:56.1200000	12:34:56.1230000	12:34:56.1235000

T5	T6	T7	T
12:34:56.1234600	12:34:56.1234570	12:34:56.1234567	12:34:56.1234567

Note Not specifying a precision is equivalent to a precision of 7.

Because you can specify precision, the size of these data types can vary, from *n* bytes to *n*+2 bytes, where *n* is the smallest size the type can be (which depends on the specific type). For example, for the *TIME* data type, *n* = 3. The following table shows how the precision affects the size of the time data type.

Precision	Size in bytes
0, 1, 2	*n*
3, 4	*n*+1
5, 6, 7	*n*+2

New Data and Time Functions and Functionality

With these new date and time data types comes some new functions and functionality. For example, the *DATEPART* and *DATENAME* functions now support the following new datepart argument values:

- **microsecond** The number of seconds to six decimal places
- **nanosecond** The number of seconds to seven decimal places, although nine are displayed (the last two are always 00)
- **TZoffset** Signed time zone offset in minutes
- **ISO_WEEK** Week number based on ISO 8601 standard where the first day of the week is Monday, and week one always contains the first Thursday and contains from four to seven days of that year (it can contain days from a previous year)

DATEADD and *DATEDIFF* can also use the microsecond and nanosecond for the datepart argument. *DATEDIFF*, however, still returns an *INT* data type, so if the difference is outside the range of -2,147,483,648 to +2,147,483,647, it will raise an exception, as shown here:

```
DECLARE
    @d1 DATETIME2 = '2001-01-01',
    @d2 DATETIME2 = '2009-01-01'

SELECT DATEDIFF(nanosecond, @d1, @d2) AS Diff
```

This query results in the following exception:

```
Msg 535, Level 16, State 0, Line 4



The datediff function resulted in an overflow. The number of dateparts separating two
date/time instances is too large. Try to use datediff with a less precise datepart.
```

Similarly, *DATEADD* can only take an *INT* data type for the amount to add, so you are limited to adding -2,147,483,648 to +2,147,483,647 of any datepart.

SWITCHOFFSET is another new function that is used to change the time zone of the input value to a specified new time zone; it can also be used to report the local *DATETIME* value of the input *DATETIMEOFFSET* value.

```
DECLARE @d1 DATETIMEOFFSET = '2008-01-01 12:34:56.1234567-06:00'
SELECT
    @d1 AS DallasTime,
    SWITCHOFFSET(@d1, '+00:00') AS LondonTime,
    SWITCHOFFSET(@d1, '-08:00') AS RedmondTime
```

This query returns the following data:

DallasTime	LondonTime	RedmondTime
1/1/2008 12:34:56 PM -06:00	1/1/2008 6:34:56 PM +00:00	1/1/2008 10:34:56 AM -08:00

Using *SWITCHOFFSET* in conjunction with *CAST* or *CONVERT* can return the local time without offset information. For example,

```
SELECT SYSDATETIME() AS [Dallas]
    , SYSDATETIMEOFFSET() AS [DallasWithOffset]
    , CAST(SWITCHOFFSET(SYSDATETIMEOFFSET(), '-08:00') AS DATETIME2) AS [Seattle]
    , SWITCHOFFSET(SYSDATETIMEOFFSET(), '-08:00') AS [SeattleWithOffset]
```

This query returns the following data:

Dallas	DallasWithOffset	Seattle	SeattleWithOffset
2008-03-04 16:03:00.777	3/4/2008 4:03:00 PM -06:00	2008-03-04 14:03:00.777	3/4/2008 2:03:00 PM -08:00

Because I am running this code on a machine in GMT -06:00, the time in Seattle should be and is two hours earlier than the Dallas times. The *TODATETIMEOFFSET* function can also return a *DATETIMEOFFSET* value but works differently from *SWITCHOFFSET*. Examine the following code and results.

```
SELECT
    SWITCHOFFSET('2008-03-04 16:16:17.162 -06:00', '-08:00') AS [Seattle1]
    , TODATETIMEOFFSET('2008-03-04 16:16:17.162 -06:00', '-08:00') AS [Seattle2]
```

This results in the following:

SeattleWithOffset1	SeattleWithOffset2
3/4/2008 2:16:17 PM -08:00	3/4/2008 4:16:17 PM -08:00

Both functions return a value of *DATETIMEOFFSET* data type, and although the same input was (apparently) used for both, their return values still differ. As shown earlier in this section, *SWITCHOFFSET* moved the time zone to one that is two hours earlier. *TODATETIMEOFFSET* does not shift time zones. It takes *DATETIME2* data as its input and does not change the time zone of the input; rather, it sets the new time zone on the *DATETIME2* input (which has no time zone). The *DATETIMEOFFSET* value that is being input into the *TODATETIMEOFFSET* function is actually being implicitly converted to *DATETIME2* before setting the time zone. That means, in this following query, all three of these returned values would be the same (3/4/2008 4:16:17 PM -08:00).

```
SELECT
  TODATETIMEOFFSET('2008-03-04 16:16:17.162 -06:00', '-08:00')
  , TODATETIMEOFFSET('2008-03-04 16:16:17.162', '-08:00')
  , TODATETIMEOFFSET(CAST('2008-03-04 16:16:17.162 -06:00' as DATETIME2), '-08:00')
```

Here are a few more notable functions of which you should be aware:

Function	Description	Example Result
SYSDATETIME	Current date and time as *DATETIME2* data type	2008-03-04 15:48:47.610
SYSDATETIMEOFFSET	Current date and time as *DATETIME2* in UTC	3/4/2008 3:48:47 PM -06:00
SYSUTCDATETIME	Current date and time as *DATETIMEOFFSET* using system time zone	2008-03-04 21:48:47.610

Again, because I happened to be in GMT -06:00 when I executed these functions, *SYSDATETIME* returned a value six hours earlier than *SYSUTCDATETIME*. *SYSDATETIMEOFFSET* returned the local time (same as *SYSDATETIME*) but with the time zone offset (in this case, -06:00).

More Info There are other functions for which you can find plenty of documentation in SQL Server 2008 Books Online under the topic "Date and Time Data Types and Functions (Transact-SQL)."

Notes on Conversion

When implicitly or explicitly converting *DATE*, *TIME*, *SMALLDATETIME*, *DATETIME*, *DATETIME2*, or string literals in a valid *DATE*, *TIME*, or *DATETIME* format (without time zone) to *DATETIMEOFFSET*, the resulting value always has a time zone of 00:00, as shown here (all three return values are 3/4/2008 4:34:05 PM +00:00).

```
DECLARE @dt DATETIME = '2008-03-04 16:34:05'
DECLARE
  @dto1 DATETIMEOFFSET = @dt
  , @dto2 DATETIMEOFFSET = CAST(@dt AS DATETIMEOFFSET)
  , @dto3 DATETIMEOFFSET = TODATETIMEOFFSET(@dt, '+00:00')
SELECT @dto1, @dto2, @dto3
```

When using *CONVERT* to convert from *DATETIMEOFFSET* to *DATE*, *TIME*, *DATETIME2*, *DATETIME*, or *SMALLDATETIME*, specifying style 0 (or no style because 0 is the default) will result in the date and time in the local format of the preserved time zone (this also applies to using CAST). Using style 1, however, will result in UTC format. For example, examine the following query and results.

```
DECLARE @dto datetimeoffset = '2008-03-06 13:45:00.1234567 -06:00'
SELECT CAST(@dto As DATETIME2) AS DT2_Cast
    , CONVERT(DATETIME2, @dto) AS DT2_NoStyle
    , CONVERT(DATETIME2, @dto, 0) AS DT2_Style0
    , CONVERT(DATETIME2, @dto, 1) AS DT2_Style1
```

DT2_Cast	DT2_NoStyle	DT2_Style0	DT2_Style1
2008-03-06 13:45:00.123	2008-03-06 13:45:00.123	2008-03-06 13:45:00.123	2008-03-06 19:45:00.123

Because the results using style 1 are being output in UTC, and because the input value is GMT 06:00, a shift of six hours in the value results. The results seen here from using style 0

and 1 apply only when converting from *DATETIMEOFFSET* to another date or time data type. Using the *CONVERT* styles when converting date and time data to character data results in the more familiar variety of date and time formats including various American National Standards Institute (ANSI), international, Open Database Connectivity (ODBC) canonical, and International Organization for Standardization (ISO) formats. See the Cast and Convert topic in SQL Server 2008 Books Online for more details.

There is one last item of note when using CONVERT and style 1. Examine the following.

```
DECLARE @dto datetimeoffset = '2008-03-06 19:45:00.1234567 -06:00'
SELECT CONVERT(DATE, @dto, 1) AS DT2_Style1
```

What do you think the result will be? If you said '2008-03-07', you are correct! The reason for this behavior is the shift to UTC. If this had been converted to *DATETIME2*, the result would have been '2008-03-07 01:45:00.123'. So, when converting to just a date data type, we still see the date change, even though the time is not part of the return value.

User-Defined Table Types and Table-Valued Parameters

User-Defined Table Type

You can now declare a type that represents a table structure. For example, suppose you frequently need to use the following structure in your T-SQL code:

```
DECLARE @NewCustomer TABLE
(
    [CustomerID] int NULL,
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [EmailAddress] varchar(128) NULL,
    [Phone] varchar(20) NULL
)
```

After repeating this code many times in a variety of stored procedures and user-defined functions, you begin to wish for an easier way to make these declarations. And of course, there is a better way—use a user-defined table type. Here is the definition of the *NewCustomer* table:

```
CREATE TYPE [dbo].[NewCustomer] AS TABLE
(
    [CustomerID] int NULL,
```

```
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [EmailAddress] varchar(128) NULL,
    [Phone] varchar(20) NULL
)
```

Once you have defined the table type, you can simply use it when declaring variables and parameters, as shown here:

```
DECLARE @NewCustomer AS [dbo].[NewCustomer]
```

This declaration is equivalent to the DECLARE @NewCustomer TABLE code found in the beginning of this section without all the extra typing required for each subsequent declaration.

Now, of course, what would a new feature be without a few caveats? One of these is that you cannot define a column as a user-defined table type—that would be the equivalent of nesting tables, which cannot be done. There are other things that are required when using user-defined table types, such as:

- Default values are not allowed in the table definition.
- Primary Key columns must be on a persisted column that does not allow null values.
- Check constraints cannot be done on nonpersisted computed columns.
- Nonclustered indexes are not allowed. If, however, you create a (nonclustered) primary key or unique constraint (both of which are allowed), you will be creating a nonclustered index behind the scenes because SQL Server uses an index to enforce unique and primary key constraints.

Note You cannot ALTER a user-defined table type. Once created, it must be dropped and re-created to make any changes. This can become cumbersome if the user-defined table type has been used as a parameter in a stored procedure or user-defined function, because it can now also not be dropped. This means that you will need to do some planning instead of haphazardly using user-defined table types.

Table-Valued Parameters

Once you have defined the necessary user-defined table types, you can then use them in your T-SQL code, as variables (shown above) or as parameters, as shown here:

```
--A stored procedure with a TVP
CREATE PROCEDURE prCustomerInsertOrUpdate
    (@NewCustomer [dbo].[NewCustomer] READONLY)
AS
MERGE Customer AS c
USING @NewCustomer AS nc
ON c.CustomerID = nc.CustomerID
WHEN TARGET NOT MATCHED THEN
    INSERT (FirstName, LastName, EmailAddress, Phone)
    VALUES (nc.FirstName, nc.LastName, nc.EmailAddress, nc.Phone)
WHEN MATCHED THEN
    UPDATE SET
        FirstName = nc.FirstName,
        LastName = nc.LastName,
        EmailAddress = nc.EmailAddress,
        Phone = nc.Phone;
GO

--Using the stored procedure
DECLARE @newcust [dbo].[NewCustomer]

INSERT INTO @newcust (CustomerID, Firstname, LastName, Email, Phone)
VALUES (NULL, 'Peter', 'DeBetta', 'peterd@adventure-works.com', '765-555-0191')

EXEC prCustomerInsertOrUpdate @newcust
```

In this example, we have created a stored procedure that accepts a customer record as a parameter and then uses the MERGE statement to either insert or update the record accordingly. You may have noticed that the parameter uses the *READONLY* keyword. This is a requirement for table-valued parameters (TVPs), which means you cannot modify a table-valued parameter using DML (which also means that it cannot be an OUTPUT parameter).

This static state of the parameter value does offer some benefits, though, such as no need to acquire locks when initially populating from the client. They also do not cause statements to recompile. And unlike other parameter types that may contain a set of data, TVPs are strongly typed. Yes, *XML* can also be strongly typed, but strongly typed *XML* is generally less efficient to process when compared with the table-valued parameter.

Other advantages include:

- Passing in *XML*, many parameter values, or a delimited parameter value (and splitting it) requires more complicated code than the code that uses a table-valued parameter.
- From client code, making multiple calls to a stored procedure with multiple parameters is much less efficient than sending all the data in a table-valued parameter in a single call and using a set-based operation against all incoming rows.

However, TVPs do have some minor restrictions. For example, statistics are not maintained for TVP columns. TVPs also have the same set of restrictions that table variables have,

such that you cannot SELECT INTO them, and you cannot use them with an INSERT EXEC statement.

When sending one or a few rows to be used by a procedure or user-defined function, table-valued parameters are a great choice and, with few exceptions, the best for performance. But when are the best times to use a TVP versus other features such as BULK INSERT?

When calling from a remote client (middle tier or rich client), more often than not, using a TVP is the best choice, with one exception: If you need to insert many, many rows, such as tens of thousands of rows (this number is not exact and should be used as a guideline and not a strict metric value) and you are directly inserting the data and not performing any complex business logic, then you should consider using BULK INSERT instead of TVPs (although the performance difference is minimal, even when inserting one hundred thousand rows). Otherwise, use table-valued parameters.

More Info See SQL Server Books Online for more details about bulk loading data into SQL Server.

Some performance benefits of TVPs include:

- It can reduce the number of roundtrips to the server.
  - Potentially unbounded data can be sent in a TVP.
- Streamed client-side interfaces increase its efficiency.
  - Efficient data transport—tabular data stream (TDS) layer has been optimized to stream large amount of data or TVPs.
  - New client-side flags for specifying presorted/unique data. For example, if the client data is already presorted, you can specify that the data is presorted through SNAC/SQLClient interfaces, and when the server sees the data is presorted, it optimizes by not sorting this data again on the server side.

And I have to mention one last benefit of using TVPs from a client: It's easy! In the next section, I'll demonstrate using TVPs from both T-SQL and C# clients.

Table-Valued Parameters in Action

The first thing that needs to be done is to set up a table, a table type, and a stored procedure that will insert or update into the new table.

```
USE AdventureWorks;
GO
DROP TABLE [dbo].[Customer];
```

```
DROP PROCEDURE prCustomerInsertOrUpdate;
DROP TYPE [dbo].[NewCustomer];
GO

CREATE TABLE [dbo].[Customer]
(
    [CustomerID] int IDENTITY(1, 1) PRIMARY KEY,
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [EmailAddress] varchar(128) NULL,
    [Phone] varchar(20) NULL
);
GO

SET IDENTITY_INSERT [dbo].[Customer] ON

INSERT INTO [dbo].[Customer] (CustomerID, FirstName, LastName, EmailAddress, Phone)
SELECT ContactID, FirstName, LastName, EmailAddress, Phone
FROM AdventureWorks.Person.Contact;

SET IDENTITY_INSERT [dbo].[Customer] OFF
GO

CREATE TYPE [dbo].[NewCustomer] AS TABLE
(
    [CustomerID] int NULL,
    [FirstName] varchar(50) NOT NULL,
    [LastName] varchar(50) NOT NULL,
    [EmailAddress] varchar(128) NULL,
    [Phone] varchar(20) NULL
);
GO

CREATE PROCEDURE prCustomerInsertOrUpdate
    (@NewCustomer [dbo].[NewCustomer] READONLY)
AS
MERGE Customer AS c
USING @NewCustomer AS nc
ON c.CustomerID = nc.CustomerID
WHEN TARGET NOT MATCHED THEN
    INSERT (FirstName, LastName, EmailAddress, Phone)
    VALUES (nc.FirstName, nc.LastName, nc.EmailAddress, nc.Phone)
WHEN MATCHED THEN
    UPDATE SET
        FirstName = nc.FirstName,
        LastName = nc.LastName,
        EmailAddress = nc.EmailAddress,
        Phone = nc.Phone;
GO
```

Some of this code is a rehash of some code earlier in this section with some new code to create and populate a sample target table. So how do you call the stored procedure from T-SQL? You first create a variable of the table type, add some rows to that variable, and then

pass it into the stored procedure in the same way you would pass a scalar value, such as an integer.

```
DECLARE @C AS [dbo].[NewCustomer];

INSERT INTO @C
VALUES
    (1, 'Gustavo', 'Achong', 'gustavo.achong@adventure-works.com', '398-555-0132')
  , (NULL, 'Peter', 'DeBetta', 'peterd@adventure-works.com', '987-555-0191');

EXEC prCustomerInsertOrUpdate @C;

SELECT *
FROM dbo.Customer
WHERE LastName IN ('Achong', 'DeBetta');
```

The last SELECT statement will return the following results, showing how one row was added and one row was updated (Gustavo's email was changed from gustavo0@adventure-works.com).

CustomerID	FirstName	LastName	EmailAddress	Phone
1	Gustavo	Achong	gustavo.achong@adventure-works.com	398-555-0132
19978	Peter	DeBetta	peterd@adventure-works.com	987-555-0191

It's just that simple. What about making the same call from a .NET client? I'll let the code speak for itself.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.SqlClient;
using System.Data;
using Microsoft.SqlServer.Server;

namespace TVPClient
{
  class Program
  {
    static void Main(string[] args)
    {
      using (SqlConnection connection = new SqlConnection())
      {
        connection.ConnectionString = @"Data Source=(local);"
                + @"Initial Catalog=adventureworks; Integrated Security=SSPI;";
        DataTable newCustomer = NewCustomer();
             SqlCommand insertPeople =
                 new SqlCommand("dbo.prCustomerInsertOrUpdate", connection);
        insertPeople.CommandType = CommandType.StoredProcedure;
```

```
        SqlParameter tvpCust = insertPeople.CreateParameter();
        tvpCust.ParameterName = "@NewCustomer";
        tvpCust.Direction = ParameterDirection.Input;
        tvpCust.SqlDbType = SqlDbType.Structured;
        tvpCust.Value = newCustomer;
        tvpCust.TypeName = "dbo.NewCustomer";
        insertPeople.Parameters.Add(tvpCust);

        connection.Open();
        insertPeople.ExecuteNonQuery();
      }
    }

    private static DataTable NewCustomer()
    {
      DataTable dt = new DataTable("NewCustomer");
      dt.Columns.Add("CustomerID", System.Type.GetType("System.Int32"));
      dt.Columns.Add("FirstName", System.Type.GetType("System.String"));
      dt.Columns.Add("LastName", System.Type.GetType("System.String"));
      dt.Columns.Add("EmailAddress", System.Type.GetType("System.String"));
      dt.Columns.Add("Phone", System.Type.GetType("System.String"));

      dt.Rows.Add(CreateRow(dt, 19978, "Peter", "DeBetta",
          "peterd@adventure-works.com", "987-555-0111"));
      dt.Rows.Add(CreateRow(dt, null, "Adam", "Machanic",
          "adama@adventure-works.com", "789-555-0111"));

      return dt;
    }
    private static DataRow CreateRow(DataTable dt, Nullable<Int32> CustomerID,
        String FirstName, String LastName, String EmailAddress, String Phone)
    {
      DataRow dr = dt.NewRow();
      dr["CustomerID"] = CustomerID == null? -1 : CustomerID;
      dr["FirstName"] = FirstName;
      dr["LastName"] = LastName;
      dr["EmailAddress"] = EmailAddress;
      dr["Phone"] = Phone;
      return dr;
    }
  }
}
```

The code in the *NewCustomer* and *CreateRow* static methods are simply creating and populating a *DataTable*. Looking at the static *Main* method, the only difference between this and a call to a stored procedure with a scalar-value parameter is found in these three lines.

```
tvpCust.SqlDbType = SqlDbType.Structured;
tvpCust.Value = newCustomer;
tvpCust.TypeName = "dbo.NewCustomer";
```

SqlDbType.Structured tells the parameter it will be sending a table-valued parameter. The *TypeName* specifies the table type name. If it is incorrectly specified, you will get an exception. And the value of the parameter is actually a *DataTable*. In addition to a *DataTable*, you can also pass a *List<SqlDataRecord>* or a *DataReader*.

So when you now execute this T-SQL code.

```
SELECT *
FROM dbo.Customer
WHERE LastName IN ('DeBetta', 'Machanic')
```

You will see these results.

CustomerID	FirstName	LastName	EmailAddress	Phone
19978	Peter	DeBetta	peterd@adventure-works.com	987-555-0111
19979	Ezio	Alboni	ezioa@adventure-works.com	789-555-0111

Summary

I cannot begin to tell you how exciting these new type system features are. The new lax validation ability in *XML* has already allowed me to solve a major *XML* schema-related problem I was having. I have begun to write an application that uses spatial data and Microsoft's Virtual Earth. I am working on a complex hierarchy model using the HIERARCHYID data type. I know several customers who will immediately benefit from table-valued parameters. The only problem I will have with all of these new features is finding the time to use them all!

Chapter 4

Programmability

SQL Server 2005 implemented a number of new features and enhancements for Transact-SQL (T-SQL). Like its predecessor, SQL Server 2008 again brings more new features and enhancement for T-SQL.

This chapter is going to cover a couple of key development areas. The initial focus will be on the new and enhanced T-SQL features, which will then lead into the common language runtime (CLR) coding enhancements, as well as some new user interface features in SQL Server Management Studio (SSMS) that relate to programmability.

Variable Declaration and Assignment

A long time ago, I learned that T-SQL was not like other programming languages. For a long time, I accepted that when I declared a variable, all I could do was to declare it. Sure, stored procedure and user-defined function parameters could be declared and set to a default, but that simply didn't happen for inline variable declarations in T-SQL. That is, until now.

Indeed, it is true. You can now declare and assign a variable in one statement. So instead of writing this code:

```
DECLARE
    @someIntVal INT,
    @someStringVal varchar(100)
SET @someIntVal = 1
SET @someStringVal = 'One'
```

You can now write this code:

```
DECLARE
    @someIntVal INT = 1,
    @someStringVal varchar(100) = 'One'
```

I know it seems like such a simple change, but believe me, it adds up over time. For example, look at this set of variable declarations and assignments from the aspnet_Membership_CreateUser stored procedure:

```
DECLARE @ApplicationId uniqueidentifier
SELECT  @ApplicationId = NULL

DECLARE @NewUserId uniqueidentifier
SELECT @NewUserId = NULL

DECLARE @IsLockedOut bit
SET @IsLockedOut = 0

DECLARE @LastLockoutDate  datetime
SET @LastLockoutDate = CONVERT( datetime, '17540101', 112 )

DECLARE @FailedPasswordAttemptCount int
SET @FailedPasswordAttemptCount = 0

DECLARE @FailedPasswordAttemptWindowStart  datetime
SET @FailedPasswordAttemptWindowStart = CONVERT(datetime, '17540101', 112)

DECLARE @FailedPasswordAnswerAttemptCount int
SET @FailedPasswordAnswerAttemptCount = 0

DECLARE @FailedPasswordAnswerAttemptWindowStart  datetime
SET @FailedPasswordAnswerAttemptWindowStart = CONVERT(datetime, '17540101', 112)

DECLARE @NewUserCreated bit

DECLARE @ReturnValue   int
SET @ReturnValue = 0

DECLARE @ErrorCode     int
SET @ErrorCode = 0

DECLARE @TranStarted   bit
SET @TranStarted = 0
```

This preceding approach allows you to see each variable declaration and assignment together (a stylistic choice). Particularly for such coding techniques, this new declare and assign ability in Microsoft SQL Server 2008 has great potential, as shown here.

```
DECLARE
    @ApplicationId uniqueidentifier = NULL,
    @NewUserId uniqueidentifier = NULL,
    @IsLockedOut bit = 0,
    @LastLockoutDate datetime = CONVERT(datetime, '17540101', 112),
    @FailedPasswordAttemptCount int = 0,
    @FailedPasswordAttemptWindowStart datetime
        = CONVERT(datetime, '17540101', 112),
    @FailedPasswordAnswerAttemptCount int = 0,
    @FailedPasswordAnswerAttemptWindowStart datetime
        = CONVERT(datetime, '17540101', 112),
    @NewUserCreated bit,
```

```
    @ReturnValue int = 0,
    @ErrorCode int = 0,
    @TranStarted bit = 0
```

You can also declare and assign using other scalar values, such as those from variables, as shown in the following example:

```
DECLARE @x int = 10
DECLARE @y int = @x + 1
```

But mind you, because the declaration of @y references @x, it cannot be in the same declaration, as shown here:

```
-- Not allowed
DECLARE @x int = 10, @y int = @x + 1
```

I expect that you are now speechless, so I will show you the second new feature for variable assignment in SQL Server 2008. I give you compound assignment:

```
DECLARE @n int = 10
SET @n += 1
SELECT @n AS [n]
```

As you expect, this will return a value of 11. Compound assignment allows you to skip the repetition of a variable that is being assigned based on itself such that SET @x = @x + 1 is the same as SET @x += 1. There is a total of eight compound assignment operators.

Operator	Description	Operator	Description
+=	Add and assign	-=	Subtract and assign
*=	Multiply and assign	/=	Divide and assign
%=	Modulo and assign	&=	Bitwise AND and assign
^=	Bitwise XOR and assign	|=	Bitwise OR and assign

Compound assignment is like any other form of assignment and cannot be done inline in a statement. So, statements such as the following are not allowed.

```
DECLARE @a int = 3
--This line of code will fail
SET @a *= @a += 6
```

You can, however, assign the same variable multiple times using a SELECT statement, as shown here.

```
DECLARE @a int = 5;
SELECT @a *= @a, @a += @a;
SELECT @a;
```

You must be asking yourself "what is the final value of @a?" As you would expect, the result is 50. First the compound multiplication is executed, changing @a value to 25 (5 * 5) and then it is added to itself, resulting in 50 (25 + 25). As a matter of fact, when you have multiple assignments in a SELECT statement, they seem to always be done from left to right, regardless of the operation being performed. This makes perfect sense being that there is no order of operations involved because these are individual assignments. So although you might be inclined to think that the following code would also return a value of 50, it actually returns 100.

```
DECLARE @a int = 5;
SELECT @a += @a, @a *= @a;
SELECT @a;
```

Just in case this isn't clear why, keep in mind that that code is the same as this code.

```
DECLARE @a int = 5;
SELECT @a = @a + @a, @a = @a * @a;
SELECT @a;
```

Assignment occurs from left to right, regardless of what actual expression is being used to determine the resulting value. What I am trying to convey is that you should not confuse compound assignment with other T-SQL operators—these are simply still assigning values and do not have an order of operation.

Table Value Constructor Through VALUE Clause

Here is an excerpt of a script that installs the msdb system database.

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 0, 1, 1, N'[Uncategorized (Local)]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 1, 1, 1, N'Jobs from MSX')
```

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 2, 1, 2, N'[Uncategorized (Multi-Server)]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 3, 1, 1, N'Database Maintenance')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 4, 1, 1, N'Web Assistant')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 5, 1, 1, N'Full-Text')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 6, 1, 1, N'Log Shipping')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES ( 7, 1, 1, N'Database Engine Tuning Advisor')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES (98, 2, 3, N'[Uncategorized]')
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
    VALUES (99, 3, 3, N'[Uncategorized]')
```

Some people prefer to use a SELECT with UNION ALL to do the same operation, as shown here:

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
SELECT  0, 1, 1, N'[Uncategorized (Local)]' UNION ALL
SELECT  1, 1, 1, N'Jobs from MSX' UNION ALL
SELECT  2, 1, 2, N'[Uncategorized (Multi-Server)]' UNION ALL
SELECT  3, 1, 1, N'Database Maintenance' UNION ALL
SELECT  4, 1, 1, N'Web Assistant' UNION ALL
SELECT  5, 1, 1, N'Full-Text' UNION ALL
SELECT  6, 1, 1, N'Log Shipping' UNION ALL
SELECT  7, 1, 1, N'Database Engine Tuning Advisor' UNION ALL
SELECT 98, 2, 3, N'[Uncategorized]' UNION ALL
SELECT 99, 3, 3, N'[Uncategorized]'
```

And now, in SQL Server 2008, you can simply execute the following statement to achieve the same results.

```
INSERT INTO dbo.syscategories (category_id, category_class, category_type, name)
VALUES
    ( 0, 1, 1, N'[Uncategorized (Local)]'),
    ( 1, 1, 1, N'Jobs from MSX'),
    ( 2, 1, 2, N'[Uncategorized (Multi-Server)]'),
    ( 3, 1, 1, N'Database Maintenance'),
    ( 4, 1, 1, N'Web Assistant'),
    ( 5, 1, 1, N'Full-Text'),
    ( 6, 1, 1, N'Log Shipping'),
    ( 7, 1, 1, N'Database Engine Tuning Advisor'),
    (98, 2, 3, N'[Uncategorized]'),
    (99, 3, 3, N'[Uncategorized]')
```

The difference is simply amazing. I was a big fan of using SELECT statements with UNION ALL to generate a row set, and now it is very apparent to me that this table value constructor (TVC) through VALUE clause feature will further simplify the scripting of data, the creation of demos, and so on. Of course, you can also use the TVC to do things besides inserting multiple rows of data. Back in an earlier chapter, I was querying the Declarative Management Framework system tables and views, as shown here:

```
;WITH AutomatedPolicyExecutionMode (ModeId, ModeName)
AS
(SELECT *
 FROM (VALUES
         (0, 'On demand'),
         (1, 'Enforce Compliance'),
         (2, 'Check on change and log'),
         (4, 'Check on schedule and log')
 ) AS EM(ModeId, ModeName)
)
SELECT
    pmf.[management_facet_id] AS FacetID
    , pmf.[name] AS FacetName
    , APEM.[ModeName]
FROM syspolicy_management_facets AS pmf
    INNER JOIN AutomatedPolicyExecutionMode AS APEM
        ON pmf.[execution_mode] & APEM.[ModeId] = APEM.[ModeId]
ORDER BY pmf.[name], APEM.[ModeId]
```

Because there was no metadata for the various execution modes, I created it using the row constructor ability combined with a common table expression (CTE) to allow me to use the TVC data in a join with another table. Again, this could have been done using a SELECT statement with UNION ALL, and it behaves in exactly the same way. The only difference is that as the number of generated rows grows, this syntax will require less typing and less space in script.

Merge

Back before SQL Server 2005 was released, there was talk of a feature that was known by the name "Merge" or "Upsert," which was the ability to insert and update a table's data in a single data manipulation language (DML) statement. Alas, the feature didn't make it into the release to manufacturing (RTM) of SQL Server 2005. Much to my satisfaction and pleasure, however, it has made it into SQL Server 2008.

The new MERGE statement has the ability to insert new data into a target table and update or delete existing data in that target table, directly or by means of a view, using a single T-SQL statement.

The abridged syntax of MERGE is as follows:

```
MERGE <target_table> [ AS table_alias ]
USING <table_source>
ON <search_condition>
[WHEN MATCHED [ AND <search_condition>]
        THEN {UPDATE… | DELETE} ]
[WHEN NOT MATCHED BY TARGET [ AND <search_condition>]
        THEN INSERT… ]
[WHEN NOT MATCHED BY SOURCE [ AND <search_condition>]
        THEN {UPDATE… | DELETE} ]
;
```

Note Although using a semicolon as a statement terminator is not officially required by most T-SQL statements, the MERGE statement does require its use.

The MERGE statement initially specifies a target table, which can actually be a table or a view (see "Modifying Data Through a View" in SQL Server 2008 Books Online for details about modifying through a view). The USING clause specifies the source of the data against which you will be comparing the target data. The ON clause specifies the basis for the comparison between the target and the source.

For example, you may be using the Customer table as your target and the NewCustomer table as the source, where NewCustomer contains a batch of customers that need to be inserted or potentially updated in your existing Customer table data. If the customer needs to be updated, the primary key value of the customer, CustomerID, is used to match the updated customer data to the existing customer, as shown here:

```
MERGE Customer AS c
USING NewCustomer AS nc
ON c.CustomerID = nc.CustomerID
WHEN NOT MATCHED BY TARGET THEN
    INSERT (FirstName, LastName, CompanyName, EmailAddress, Phone)
    VALUES (nc.FirstName, nc.LastName, nc.CompanyName, nc.EmailAddress,
            nc.Phone)
WHEN MATCHED THEN
    UPDATE SET
        FirstName = nc.FirstName,
        LastName = nc.LastName,
        CompanyName = nc.CompanyName,
        EmailAddress = nc.EmailAddress,
        Phone = nc.Phone;
```

In this preceding example, new customers (which would have a CustomerID of 0 or NULL) would get inserted into the Customer table, and existing customers (matched on CustomerID) would always be updated, regardless whether the data in the target and source was different.

The WHEN Clauses

The key to MERGE are the three WHEN clauses used to determine when the various actions should take place: WHEN MATCHED, WHEN NOT MATCHED BY TARGET, and WHEN NOT MATCHED BY SOURCE.

WHEN MATCHED

The WHEN MATCHED clause is used to find matches between the target and source tables and can either perform an UPDATE or DELETE against the target table. There are a few rules you need to keep in mind when using this clause.

- When performing an UPDATE with this clause, the match condition must only return one row from the source table because the MERGE statement cannot perform the same DML operation on a single row of data in the target more than one time. If the WHEN MATCHED condition returns more than one row and an UPDATE is used, an error will result.
- WHEN MATCHED can be used, at most, two times in the MERGE statement. If used twice, the following rules also apply:
  - The two clauses are processed in order.
  - One clause must UPDATE, and the other one must DELETE (order is not important).
  - The second WHEN MATCH clause is checked only if the first is not satisfied.
  - The first WHEN MATCHED clauses must specify additional criteria. If you attempt to execute without specifying additional search criteria for the first WHEN MATCHED clause, you will receive an error.

Note Although the first clause is required to have additional criteria, it doesn't prevent you from using the same criteria for both WHEN MATCHED clauses. If you do use the same criteria for both WHEN MATCHED clauses, the second one will never get processed because the second clause only gets checked when the first one does not match the specified criteria. If they have the same criteria, the second would also not match.

WHEN NOT MATCHED [BY TARGET]

WHEN NOT MATCHED BY TARGET (BY TARGET is optional, although I suggest being explicit and using it) is used to determine if there are rows in the source table that don't exist in the target table. In other words, it is used to find rows that might need to be inserted into the target table from the source table.

- This is the only WHEN clause that can INSERT data into the source table.
- There can be, at most, one WHEN NOT MATCHED BY TARGET clause in a MERGE statement.
- Additional criteria is optional.

WHEN NOT MATCHED BY SOURCE

This clause is used to find rows in the target table that do not exist in the source table. Here are some rules to abide by when using WHEN NOT MATCHED BY SOURCE:

- WHEN NOT MATCHED BY SOURCE can be used, at most, two times in the MERGE statement. If used twice, the following rules apply:
  - The two clauses are processed in order.
  - One clause must UPDATE, and the other one must DELETE (order is not important).
  - The second clause is checked only if the first is not satisfied.
  - The first clause must specify additional criteria. If you attempt to execute without specifying additional search criteria for the first clause, you will receive an error.

Note Although the first clause is required to have additional criteria, it doesn't prevent you from using the same criteria for both WHEN NOT MATCHED BY SOURCE clauses. If you do use the same criteria for both WHEN NOT MATCHED BY SOURCE clauses, the second one will never get processed because the second clause only gets checked when the first one does not match the specified criteria. If they have the same criteria, the second would also not match.

Other Notes on All Matching Clauses

INSTEAD OF triggers defined on a target table work in the same fashion as they always have. They do not actually modify the underlying table but rather rely on the INSTEAD OF trigger to do that work (by the now populated *inserted* and *deleted* trigger tables). Also, if you define

an INSTEAD OF trigger for any action on the target table, there must be an INSTEAD OF trigger for all actions on the target table.

Note Triggers only see the rows affected by the associated trigger action. So an insert trigger only sees inserted rows (and not the deleted or updated ones). An update trigger only sees updated rows, and a delete trigger only sees deleted rows.

Also, because each trigger is called for each action, a trigger that handles multiple actions can be fired once for each action (assuming all actions occurred). For example, if the MERGE statement inserts and updates rows and there is a trigger defined for insert and update, that trigger will be called twice—once for the insert and once for the update, although not in any particular order.

To achieve the correct matching, the MERGE statement does outer joins between the target and source data as needed. When using the WHEN NOT MATCHED BY TARGET clause, an outer join occurs from the source table to the target table. This means that, for all rows from the source table, MERGE checks to see if a match exists in the target, which means that all rows from the source are returned. The converse is true when using the WHEN NOT MATCHED BY SOURCE clause, and all rows from the target table are returned in order to perform an outer join from the target table to the source table.

Therefore, if you use both the WHEN NOT MATCHED BY TARGET and the WHEN NOT MATCHED BY SOURCE clauses in a single MERGE statement, an outer join is used in both directions resulting in the use of a full outer join.

MERGE Exemplified

Let's go through a simple example using the new AdventureWorksLT database sample. This new sample database contains customers and their associated sales data. For reporting purposes, I want to create a flat view of customers and sales total and last sales date, so I first create a table to hold that information, as shown here:

```
USE AdventureWorksLT;
GO
CREATE TABLE SalesLT.CustomerTotals
(
    CustomerID int PRIMARY KEY,
    LastOrderDate datetime,
    SalesTotal money
);
GO
```

From here, I will do a very simple merge between the Customer table and the new CustomerTotals table.

```
MERGE SalesLT.CustomerTotals AS ct
USING SalesLT.Customer AS c
ON c.CustomerID = ct.CustomerID
WHEN NOT MATCHED BY TARGET THEN
    INSERT (CustomerID)
        VALUES (c.CustomerID);
```

This MERGE statement looks to see what rows in Customer (the source) do not have a corresponding match in CustomerTotals (the target). Because the CustomerTotal table is empty, all rows in Customer should meet the criteria, and you should see a message that 440 rows were affected (assuming you haven't deleted any customers yet). Running it a second time will result in zero (0) rows being affected, because all rows will now match between target and source (because you just inserted them from the source to the target).

Now, you will use data from the SalesOrderHeader table to update the CustomerTotals table with sales total and last order date data, as shown here:

```
WITH CustSales AS
(   SELECT
        CustomerID,
        MAX(OrderDate) as MaxOrderDate,
        SUM(TotalDue) as TotalDueTotal
    FROM SalesLT.SalesOrderHeader
    GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING CustSales
ON CustSales.CustomerID = ct.CustomerID
WHEN MATCHED THEN
    UPDATE SET
        LastOrderDate = CustSales.MaxOrderDate,
        SalesTotal = CustSales.TotalDueTotal
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
WHEN NOT MATCHED BY TARGET THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (CustSales.CustomerID, CustSales.MaxOrderDate, CustSales.TotalDueTotal);
```

So what is this MERGE statement actually doing? The MERGE is updating the last sales date and total sales amount for any customer that exists in both the target (CustomerTotals) and the source (a CTE of aggregated customer sales data). The MERGE is also deleting any customers from the target table that don't have any associated sales (don't exist in the customer sales CTE), and it is adding new customers into the target table if they don't already exist in that table but do exist in the source. As you may now realize, the first step to populate the CustomerTotals table was unnecessary because this statement would also populate the CustomerID values (without the overhead of populating them all and then removing a

majority of them). What you may not have noticed is that, although this MERGE seems to be exactly what you want, it is very inefficient. Stop for a moment and examine the code and see if you can spot the inefficiency. I'll wait for you…

Optimizing MERGE

Now, you have pondered for at least a few moments about the potential performance problem the last MERGE statement could cause. So where is the inefficiency? It is in the WHEN MATCHED clause. Imagine if you had millions of active customers and you ran that MERGE statement that checked the sales order header data against the customer totals data. Every time you execute that MERGE statement, you would be updating millions of records with the same data—not very efficient. A better choice would be to update only those records that have changed. And so, a small change to the WHEN MATCHED clause can make a world of difference, as shown here:

```
WITH CustSales AS
(   SELECT
        CustomerID,
        MAX(OrderDate) as MaxOrderDate,
        SUM(TotalDue) as TotalDueTotal
    FROM SalesLT.SalesOrderHeader
    GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING CustSales
ON CustSales.CustomerID = ct.CustomerID
WHEN MATCHED AND LastOrderDate != CustSales.MaxOrderDate THEN
    UPDATE SET
        LastOrderDate = CustSales.MaxOrderDate,
        SalesTotal = CustSales.TotalDueTotal
WHEN NOT MATCHED  BY SOURCE THEN
    DELETE
WHEN NOT MATCHED BY TARGET THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (CustSales.CustomerID, CustSales.MaxOrderDate, CustSales.TotalDueTotal);
```

Only if the customer IDs match and the date of the last order in the sales order data is different from the date of the last order in the customer totals data will the CustomerTotals table be updated. The point of this exercise is to make you aware that you should test your code and ensure that you are only doing the operations that you need to do to insert, update, or delete your target data.

Still another alternate approach would be to update only since the last known change (which also means you wouldn't delete missing records because your source would only have a subset of customers), as shown here:

```
WITH CustSales AS
(SELECT CustomerID, MAX(OrderDate) as MaxOrderDate, SUM(TotalDue) as TotalDueTotal
    FROM SalesLT.SalesOrderHeader
    WHERE OrderDate >
        ISNULL((SELECT MAX(LastOrderDate) FROM SalesLT.CustomerTotals), 0)
    GROUP BY CustomerID
)
MERGE SalesLT.CustomerTotals AS ct
USING CustSales
ON CustSales.CustomerID = ct.CustomerID
WHEN MATCHED THEN
    UPDATE SET
        LastOrderDate = CustSales.MaxOrderDate,
        SalesTotal += CustSales.TotalDueTotal
-- WHEN NOT MATCHED BY SOURCE THEN -- removed due to dire consequences
--    DELETE                    -- if left in (see explanation below)
WHEN NOT MATCHED BY TARGET THEN
    INSERT (CustomerID, LastOrderDate, SalesTotal)
    VALUES (CustSales.CustomerID, CustSales.MaxOrderDate, CustSales.TotalDueTotal);
```

You should take special care to note the removal of the WHEN NOT MATCHED BY SOURCE clause. If that clause had not been removed and the MERGE statement is run multiple times, it will eventually get into a two-phase cycle of an empty target table and a fully loaded target table. If the source table data had not changed since last execution of this merge, the CustSales CTE would have returned 0 rows, and because the source would have no matches, all the target table data would be deleted. Then, on the next execution, because the target would have no rows, it would be fully populated.

The moral of this story: When using MERGE, be careful when you use a source that has a different scope than the target (a CTE or view that limits, a temporary table with limited data, and so on), and always, always do thorough testing.

Indexing and MERGE

One last item of note is how indexes affect the performance of MERGE. The columns in the ON clause obviously play a big role in how well MERGE will perform. For better performance, both the target and source should have an index on the columns used in the ON clause, and, if both of those indexes are unique, you will get still better performance.

In the example above, the target table has a primary key on the ON clause column—a good thing. The source, however, is a CTE that happens to GROUP BY the column used in the MERGE statement's ON clause. Fortunately in this situation, this grouping essentially becomes the unique index for the CTE (assuming a streamed aggregator is used). This may not always be the case, and, as you saw, even slight variations in the CTE could have dire effects. Again, take care when using a CTE (or a view) as the source for a MERGE.

In the end, how you use and optimize the use of MERGE will be entirely up to what the business requires, or in consulting words—it depends.

OUTPUT and MERGE

The OUTPUT clause can also be used with the MERGE statement. Like SQL Server 2005, you still have the INSERTED and DELETED references, so you can see the inserted data; the deleted data; and, for updates, the original and changed data. But OUTPUT with MERGE offers two more special features.

The first is the special $ACTION value, which returns INSERT, UPDATE, or DELETE to indicate what action was taken on that particular row. For example, examine the following code.

```
DECLARE @T1 TABLE (id int primary key, name varchar(10))
DECLARE @T2 TABLE (id int primary key, name varchar(10))

INSERT INTO @T1
VALUES
    (1, 'A'),
    (2, 'B'),
    (3, 'C'),
    (4, 'D'),
    (5, 'E'),
    (6, 'F')


INSERT INTO @T2
VALUES
    (1, 'A'),
    (3, 'C'),
    (5, 'R'),
    (7, 'T')

MERGE @T1 AS t1
USING @T2 AS t2
ON t1.id = t2.id
WHEN MATCHED and t1.name != t2.name THEN
    UPDATE
        SET name = t2.name
WHEN NOT MATCHED BY TARGET THEN
    INSERT VALUES (t2.id, t2.name)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT $ACTION, DELETED.*, INSERTED.*;

SELECT * FROM @T1
```

This would return the following two sets of results.

$ACTION	id (deleted)	name (deleted)	id (inserted)	name (inserted)
DELETE	2	B	NULL	NULL
DELETE	4	D	NULL	NULL
UPDATE	5	E	5	R
DELETE	6	F	NULL	NULL
INSERT	NULL	NULL	7	T

Id	name
1	A
3	C
5	R
7	T

The second table shows the final values in the target table, @T1. The first table shows the results of the OUTPUT clause. It reveals that row with id 5 was updated, row with id 7 was inserted, and rows with id 2, 4, and 6 were deleted. Because the WHEN MATCHED clause excluded matches between the source and target where the name was equal, rows with ids 1 and 3 were not affected.

Now what if I needed to store those OUTPUT results in another table? Examine this code.

```
DECLARE @T1 TABLE (id int primary key, name varchar(10))
DECLARE @T2 TABLE (id int primary key, name varchar(10))
DECLARE @T3 TABLE
  ([action] nvarchar(10), oldid int, oldname varchar(10), id int, name varchar(10))

INSERT INTO @T1
VALUES
    (1, 'A'),
    (2, 'B'),
    (3, 'C'),
    (4, 'D'),
    (5, 'E'),
    (6, 'F')

INSERT INTO @T2
VALUES
    (1, 'A'),
```

```
    (3, 'C'),
    (5, 'R'),
    (7, 'T')

INSERT INTO @T3
SELECT *
FROM
(MERGE @T1 AS t1
USING @T2 AS t2
ON t1.id = t2.id
WHEN MATCHED and t1.name != t2.name THEN
    UPDATE
        SET name = t2.name
WHEN NOT MATCHED BY TARGET THEN
    INSERT VALUES (t2.id, t2.name)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT $ACTION, DELETED.id as oldid, DELETED.name as oldname, INSERTED.*
) AS tChange;
;

SELECT * FROM @T3
SELECT * FROM @T1
```

Although the results tab will show exactly the same results as the previous code example, this code is actually using the OUTPUT rows as the source for an outer INSERT statement. Now you can do more than return these resulting OUTPUT values from the MERGE to some client app; you can store changes as needed. For example, you may delete rows from the target table, but you could use these features to store a copy of just the deleted rows into another table, as shown in this example.

```
DECLARE @T1 TABLE (id int primary key, name varchar(10))
DECLARE @T2 TABLE (id int primary key, name varchar(10))
DECLARE @T1Deleted TABLE (id int, name varchar(10))

INSERT INTO @T1
VALUES
    (1, 'A'),
    (2, 'B'),
    (3, 'C'),
    (4, 'D'),
    (5, 'E'),
    (6, 'F')

INSERT INTO @T2
VALUES
    (1, 'A'),
    (3, 'C'),
    (5, 'R'),
    (7, 'T')
```

```
INSERT INTO @T1Deleted (id, name)
SELECT id, name
FROM
(MERGE @T1 AS t1
USING @T2 AS t2
ON t1.id = t2.id
WHEN MATCHED and t1.name != t2.name THEN
    UPDATE
        SET name = t2.name
WHEN NOT MATCHED BY TARGET THEN
    INSERT VALUES (t2.id, t2.name)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
OUTPUT $ACTION, DELETED.id , DELETED.name
) AS tChange ([Action], [id], [name])
WHERE [Action] = N'DELETE';
;

SELECT * FROM @T1Deleted
SELECT * FROM @T1
```

This last example adds all deleted data to the @T1Deleted table, so although three rows are removed from @T1, they are kept in @T1Deleted.

GROUP BY GROUPING SETS

Do you remember the COMPUTE [BY] clause? How about GROUP BY … WITH CUBE or GROUP BY … WITH ROLLUP? Well, times have changed, and these non-ISO-compliant clauses of the SELECT statement have been replaced with International Organization for Standardization (ISO)-compliant equivalents. In addition, the GROUP BY clause has been enhanced to allow for custom groupings that would be more difficult or cumbersome to achieve using the older syntax.

To demonstrate how these new grouping features work, I am going to use variations on the following query:

```
USE AdventureWorks;
GO
;WITH SalesData AS
(
    SELECT
        ST.[Group] AS [Region]
        , ST.CountryRegionCode AS [Country]
        , S.Name AS [Store]
        , CON.LastName AS [SalesPerson]
        , SOH.TotalDue
    FROM Sales.Customer AS C
```

```
        INNER JOIN Sales.Store AS S ON C.CustomerID  = S.CustomerID
        INNER JOIN Sales.SalesTerritory AS ST ON C.TerritoryID  = ST.TerritoryID
        INNER JOIN Sales.SalesOrderHeader AS SOH ON S.CustomerID = SOH.CustomerID
        INNER JOIN HumanResources.Employee AS E ON E.EmployeeID = SOH.SalesPersonID
        INNER JOIN Person.Contact AS CON ON CON.ContactID = E.ContactID
    WHERE ST.[Group] = N'Europe'
        AND S.Name LIKE 'O[^i]%'
)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Region, Country, Store, SalesPerson;
```

This query returns the following data:

Region	Country	Store	SalesPerson	TotalSales
Europe	DE	Off-Price Bike Center	Alberts	502.0217
Europe	DE	Off-Price Bike Center	Valdez	2076.5676
Europe	GB	Outdoor Aerobic Systems Company	Alberts	486.3925
Europe	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586

The minimal number of rows returned will make it easier to follow the coming examples. Also, for the sake of not repeating the same CTE over and over again, I will abbreviate further versions of this query as follows:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Region, Country, Store, SalesPerson;
```

Now, onto the good stuff....

GROUPING SETS

Imagine, if you will, you want to generate a set of aggregate results, but you want to group by varying columns. For example, you want totals by the following groups:

- Region, Country, Store, and SalesPerson
- Region and Country
- Country
- Region
- None (grand total)

In SQL Server 2005, you would have to use the following query to achieve such a result.

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Region, Country, Store, SalesPerson

UNION ALL

SELECT Region, Country, NULL, NULL, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Region, Country

UNION ALL

SELECT NULL, Country, NULL, NULL, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Country

UNION ALL

SELECT Region, NULL, NULL, NULL, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY Region

UNION ALL

SELECT NULL, NULL, NULL, NULL, SUM(TotalDue) AS [TotalSales]
FROM SalesData;
```

This query would produce the following results:

Region	Country	Store	SalesPerson	TotalSales
Europe	DE	Off-Price Bike Center	Alberts	502.0217
Europe	DE	Off-Price Bike Center	Valdez	2076.5676
Europe	DE	NULL	NULL	2578.5893
NULL	DE	NULL	NULL	2578.5893
Europe	GB	Outdoor Aerobic Systems Company	Alberts	486.3925
Europe	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586
Europe	GB	NULL	NULL	4374.2511
NULL	GB	NULL	NULL	4374.2511
NULL	NULL	NULL	NULL	6952.8404
Europe	NULL	NULL	NULL	6952.8404

Using the *GROUPING SETS* feature, you can now rewrite this query with the same results as follows:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY GROUPING SETS
(
    (Region, Country, Store, SalesPerson),
    (Region, Country),
    (Country),
    (Region),
    ()
);
```

Each grouping set represents the grouping that appeared in each of the SELECT statements in the version that uses UNION ALL: five SELECT statements in the first and five grouping sets in the second.

Now not only is the grouping sets version of the query far less complex and easier on the eyes, but it will perform better. The performance gain is evidenced by the fact that the total number of logical reads on all tables for the first version (using UNION ALL) is 4429, and for the second version (using GROUPING SETS), it is 1022. That's a dramatic difference. The reason for such a difference is that the first query actually executes five different queries against the same underlying data and concatenates the results, and the second performs a single query against the data.

Note These logical read numbers may vary slightly depending on what modifications you have made or not made to the AdventureWorks sample database.

ROLLUP

The GROUP BY ROLLUP clause returns the grouped items and any aggregate value and adds super-aggregates that roll up from right to left. For example, let's say you want to add some additional grouping to the query as follows:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
```

```
GROUP BY GROUPING SETS
(
    (Region, Country, Store, SalesPerson),
    (Region, Country, Store),
    (Region, Country),
    (Region),
    ()
);
```

This would give you the following results:

Region	Country	Store	SalesPerson	TotalSales
Europe	DE	Off-Price Bike Center	Alberts	502.0217
Europe	DE	Off-Price Bike Center	Valdez	2076.5676
Europe	DE	Off-Price Bike Center	NULL	2578.5893
Europe	DE	NULL	NULL	2578.5893
Europe	GB	Outdoor Aerobic Systems Company	Alberts	486.3925
Europe	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586
Europe	GB	Outdoor Aerobic Systems Company	NULL	4374.2511
Europe	GB	NULL	NULL	4374.2511
Europe	NULL	NULL	NULL	6952.8404
NULL	NULL	NULL	NULL	6952.8404

In this query, you are adding additional groupings by removing columns one at a time from the right side of the grouping. When you perform this type of grouping, it is known as a roll-up, and you can use the ROLLUP clause to get the same results with a more simple T-SQL statement.

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY ROLLUP(Region, Country, Store, SalesPerson);
```

From a performance standpoint, this query is no more efficient than the equivalent query using GROUPING SETS, because ROLLUP is a shortcut for that query that uses GROUPING SETS. It is easier on the eyes and easier to type, and it makes writing the code a lot simpler and reviewing it later on simpler. As a rule, ROLLUP is equivalent to *n* + 1 GROUPING SETS, where *n* is the number of columns in the ROLLUP clause.

CUBE

Now imagine you wanted to create a grouping of all combinations of all columns being grouped, such as the following query does:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY GROUPING SETS
(
    (Region, Country, Store, SalesPerson),
    (Region, Country, Store),
    (Region, Country, SalesPerson),
    (Region, Store, SalesPerson),
    (Country, Store, SalesPerson),
    (Region, Country),
    (Region, Store),
    (Region, SalesPerson),
    (Country, Store),
    (Country, SalesPerson),
    (Store, SalesPerson),
    (Region),
    (Country),
    (Store),
    (SalesPerson),
    ()
);
```

As you add GROUPING SETS, you increase the number of rows output, because you are adding additional aggregations of varying columns to your results, as shown here:

Region	Country	Store	SalesPerson	TotalSales
Europe	DE	Off-Price Bike Center	Alberts	502.0217
NULL	DE	Off-Price Bike Center	Alberts	502.0217
NULL	NULL	Off-Price Bike Center	Alberts	502.0217
Europe	GB	Outdoor Aerobic Systems Company	Alberts	486.3925
NULL	GB	Outdoor Aerobic Systems Company	Alberts	486.3925
NULL	NULL	Outdoor Aerobic Systems Company	Alberts	486.3925
NULL	NULL	NULL	Alberts	988.4142
Europe	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586
NULL	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586
NULL	NULL	Outdoor Aerobic Systems Company	Saraiva	3887.8586
NULL	NULL	NULL	Saraiva	3887.8586

Region	Country	Store	SalesPerson	TotalSales
Europe	DE	Off-Price Bike Center	Valdez	2076.5676
NULL	DE	Off-Price Bike Center	Valdez	2076.5676
NULL	NULL	Off-Price Bike Center	Valdez	2076.5676
NULL	NULL	NULL	Valdez	2076.5676
NULL	NULL	NULL	NULL	6952.8404
Europe	NULL	Off-Price Bike Center	Alberts	502.0217
Europe	NULL	Off-Price Bike Center	Valdez	2076.5676
Europe	NULL	Off-Price Bike Center	NULL	2578.5893
NULL	NULL	Off-Price Bike Center	NULL	2578.5893
Europe	NULL	Outdoor Aerobic Systems Company	Alberts	486.3925
Europe	NULL	Outdoor Aerobic Systems Company	Saraiva	3887.8586
Europe	NULL	Outdoor Aerobic Systems Company	NULL	4374.2511
NULL	NULL	Outdoor Aerobic Systems Company	NULL	4374.2511
Europe	DE	NULL	Alberts	502.0217
NULL	DE	NULL	Alberts	502.0217
Europe	DE	NULL	Valdez	2076.5676
NULL	DE	NULL	Valdez	2076.5676
NULL	DE	NULL	NULL	2578.5893
Europe	GB	NULL	Alberts	486.3925
NULL	GB	NULL	Alberts	486.3925
Europe	GB	NULL	Saraiva	3887.8586
NULL	GB	NULL	Saraiva	3887.8586
NULL	GB	NULL	NULL	4374.2511
Europe	NULL	NULL	Alberts	988.4142
Europe	NULL	NULL	Saraiva	3887.8586
Europe	NULL	NULL	Valdez	2076.5676
Europe	NULL	NULL	NULL	6952.8404
Europe	DE	Off-Price Bike Center	NULL	2578.5893
NULL	DE	Off-Price Bike Center	NULL	2578.5893
Europe	GB	Outdoor Aerobic Systems Company	NULL	4374.2511
NULL	GB	Outdoor Aerobic Systems Company	NULL	4374.2511
Europe	DE	NULL	NULL	2578.5893
Europe	GB	NULL	NULL	4374.2511

When you want to produce GROUPING SETS for all combinations of all columns, the formula 2^n determines how many GROUPING SETS are needed for a set of columns, where *n* repre-

sents the number of columns being grouped. In the query above, the four grouped columns gives you 2^4, or 16 GROUPING SETS. Now, wouldn't it be nice if there was some shortcut (like ROLLUP) that could produce the same results? Of course it would be, and here it is:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY CUBE (Region, Country, Store, SalesPerson);
```

Now wasn't that just way too easy?

GROUPING_ID

Now what if you need to find out to which grouping set a row corresponds? For example, in the following query, we want to know to which of the five generated grouping sets each row belongs.

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
    , GROUPING_ID(Region, Country, Store, SalesPerson) AS GroupingID
FROM SalesData
GROUP BY ROLLUP(Region, Country, Store, SalesPerson);
```

This query produces these results:

Region	Country	Store	SalesPerson	TotalSales	GroupingID
Europe	DE	Off-Price Bike Center	Alberts	502.0217	0
Europe	DE	Off-Price Bike Center	Valdez	2076.5676	0
Europe	DE	Off-Price Bike Center	NULL	2578.5893	1
Europe	DE	NULL	NULL	2578.5893	3
Europe	GB	Outdoor Aerobic Systems Company	Alberts	486.3925	0
Europe	GB	Outdoor Aerobic Systems Company	Saraiva	3887.8586	0
Europe	GB	Outdoor Aerobic Systems Company	NULL	4374.2511	1
Europe	GB	NULL	NULL	4374.2511	3
Europe	NULL	NULL	NULL	6952.8404	7
NULL	NULL	NULL	NULL	6952.8404	15

Now you may find it odd that the numbers aren't consecutive. That is because the GROUPING_ID returns based on all possible combinations. GROUPING_ID of 0 means that the grouping set result grouped by all columns. GROUPING_ID of 1 means it grouped by Region, Country, and Store (columns 1, 2 ,and 3, or all but 4). GROUPING_ID of 2 means it grouped by Region, Country, and SalesPerson (columns 1, 2, and 4, or all but 3). GROUPING_ID of 3 means it grouped by Region and Country (columns 1 and 2 only). Because this query doesn't group by all possible combinations, it will skip GROUPING_IDs it doesn't produce. CUBE will produce GROUPING_ID from 0 to $2^n - 1$, where *n* represents the number of columns being grouped by.

Miscellaneous Thoughts

If you have done any work with Analysis Services (AS) cubes, you will notice some similarities between simple AS cubes and GROUPING SETS. Although I am not proposing this as a replacement for AS cubes in general, it could be used as an alternate for simple AS cubes.

GROUPING SETS can contain multiple column lists and also multiple CUBE and ROLLUP clauses, as shown here:

```
;WITH SalesData AS
(...)
SELECT Region, Country, Store, SalesPerson, SUM(TotalDue) AS [TotalSales]
FROM SalesData
GROUP BY GROUPING SETS
(
    CUBE (Region, Country),
    CUBE (Region, Store),
    CUBE (Store, SalesPerson)
);
```

This produces 3 groups of 4 grouping sets, for a total of 12, but only 8 of them are distinct.

CUBE	GROUPING SET
CUBE (Region, County)	Region, County
CUBE (Region, County)	Region
CUBE (Region, County)	County
CUBE (Region, County)	()
CUBE (Region, Store)	Region, Store
CUBE (Region, Store)	Region
CUBE (Region, Store)	Store
CUBE (Region, Store)	()
CUBE (Store, SalesPerson)	Store, SalesPerson

CUBE	GROUPING SET
CUBE (Store, SalesPerson)	Store
CUBE (Store, SalesPerson)	SalesPerson
CUBE (Store, SalesPerson)	()

Notice that you have 4 grand totals () grouping sets, as well as repeats for Region and Store. If you had used the GROUPING_ID function, it would have revealed the repeated grouping sets.

And finally, when using GROUP BY, there is no limit on the number of expressions. When using the more complex GROUPING SETS, ROLLUP, and CUBE, however, you can have at most 32 expressions and at most 2^{12} (4,096) generated grouping sets. And because CUBE produces 2^n grouping sets, CUBE can have at most 12 columns.

Object Dependencies

Object Dependency tracking in SQL Server 2005 and earlier versions was not as reliable as people would have liked it to be. I have some exciting news for SQL Server 2008: Object dependency functionality is better than its predecessors. Three new system objects have been added to the Database Engine that you definitely want to know about:

- sys.sql_expression_dependencies
  - This is a new catalog view that is meant to replace sys.sql_dependencies.
  - It tracks both schema-bound and non-schema–bound dependencies.
  - It tracks cross-database and cross-server references. (Only the name is returned; the IDs are not resolved.)
  - Does not contain information about rules, defaults, temporary tables, temporary stored procedures, or system objects.
- sys.dm_sql_referenced_entities
  - This is a new dynamic management function that replaces sp_depends functionality.
  - It returns a row for each entity referenced by a given entity.
  - It has two arguments:
    - The referencing entity name. Schema name is also required when the referencing class is OBJECT.
    - The referencing class that can be one of these three values: *OBJECT*, *DATABASE_DDL_TRIGGER*, or *SERVER_DDL_TRIGGER*.

- sys.dm_sql_referencing_entities
  - This too is a new dynamic management function that replaces sp_depends functionality.
  - It returns a row for each entity *referencing* a given entity.
  - Two arguments:
    - The referenced entity name. Schema name is also required when the referencing class is *OBJECT*, *TYPE*, or *XML_SCHEMA_COLLECTION*.
    - The referenced class that can be one of these four values: *OBJECT*, *TYPE*, *XML_SCHEMA_COLLECTION*, or *PARTITION_FUNCTION*.

All three of these new system objects return varying table/view structures. SQL Server 2008 Books Online contains detailed information about these return structures.

CLR Enhancements

CLR integration introduced in SQL Server 2005 extended the programming choices a developer had. Although CLR integration does not replace T-SQL but rather extends it, some kinds of code, like complicated string manipulation, is better and faster when programmed using the CLR.

You can use CLR-based code to create stored procedures, user-defined functions, triggers, user-defined aggregates, and user-defined types. But there were limitations to its use, such as an 8K maximum size on user-defined types and user-defined aggregates.

Large Aggregates

A common scenario that requires a potentially lengthy string return value from an aggregate is string concatenation. In a previous incarnation, the CLR-based string concatenation function that I (and so many others) wrote was limited to a return string of no more than 8,000 bytes. This was very limiting, for even a column with an average string length of 20 (including the added comma) could concatenate no more than 400 rows before failing. So this following query (assuming you had a user-defined aggregate, or UDA, in SQL Server 2005 named dbo.Concat2005) would fail.

```
USE AdventureWorks;
GO
SELECT dbo.Concat(EmailAddress)
FROM Person.Contact;
```

But SQL Server 2008 comes to the rescue, and you can now code the MaxByteSize named parameter of the SqlUserDefinedAggregate attribute to be a value of -1, which translates to "up to 2 GB." For this example, I have included the entire UDA codebase.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;

namespace debetta
{
    [Serializable]
    [SqlUserDefinedAggregate(Format.UserDefined, MaxByteSize = -1)]
    public struct Concat: IBinarySerialize
    {
        StringBuilder _sb;
        public void Init()
        {
            _sb = new StringBuilder(System.String.Empty);
        }
        public void Accumulate(SqlString Value)
        {
            if (Value.IsNull == false && Value.Value.Trim().Length > 0)
                _sb.Append(Value.Value + ",");
        }
        public void Merge(Concat Group)
        {
            if (Group._sb.ToString().Length > 0)
                _sb.Append(Group._sb.ToString()+ ",");
        }
        public SqlString Terminate()
        {
            return new SqlString(_sb.ToString());
        }
        public void Read(System.IO.BinaryReader r)
        {
            _sb = new StringBuilder(r.ReadString());
        }
        public void Write(System.IO.BinaryWriter w)
        {
            w.Write(_sb.ToString());
        }
    }
}
```

Now our new query using our new UDA in SQL Server 2008 works:

```
USE AdventureWorks;
GO
SELECT dbo.Concat(EmailAddress)
FROM Person.Contact;
```

For the record, the resulting string is 570,812 characters long, or 1,141,624 bytes, and the query runs in about 150 milliseconds.

UDAs also offer a new ability—multiple parameter input on the Accumulate method. In other words, I can aggregate against one or more values, such as variables, or I can aggregate against multiple columns. Examine the following UDA that finds the max MONEY value in two different columns.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;

namespace debetta
{
    [Serializable]
    [SqlUserDefinedAggregate(Format.Native, IsInvariantToNulls = true,
            IsInvariantToOrder = false, IsInvariantToDuplicates = true)]
    public struct MoneyMaxDuo
    {
        public SqlMoney m_Result;

        public void Init()
        {
            m_Result = SqlMoney.Null;
        }

        public void Accumulate(SqlMoney p1, SqlMoney p2)
        {
            SqlMoney ptemp = SqlMoney.Null;

            if (p1.IsNull && p2.IsNull)
                return;
```

```
            if (p1.IsNull)
                ptemp = p2;
            else if (p2.IsNull)
                ptemp = p1;
            else
                ptemp = (p1 > p2) ? p1 : p2;

            if (m_Result.IsNull | ptemp > m_Result)
                m_Result = ptemp;

        }

        public void Merge(MoneyMaxDuo MergeObject)
        {
            if ((m_Result.IsNull) | (!MergeObject.Terminate().IsNull
                    & m_Result < MergeObject.Terminate()))
                m_Result = MergeObject.Terminate();
        }

        public SqlMoney Terminate()
        {
            return m_Result;
        }
    }
}
```

Notice that the Accumulate method has two parameters defined. When creating the aggregate in T-SQL, you must define both parameters, as shown here:

```
CREATE AGGREGATE dbo.MoneyMaxDuo (@val1 MONEY, @val2 MONEY)
RETURNS MONEY
EXTERNAL NAME MyCodeLibrary.[debetta.MoneyMaxDuo]
```

And then you can use it as follows:

```
DECLARE @T TABLE (Category CHAR(1), Price1 MONEY, Price2 MONEY)
INSERT INTO @T
VALUES ('A', 1, 2), ('A', 3, 2), ('A', 5, 7), ('B', 8, 9), ('B', 10, 6)

SELECT dbo.MoneyMaxDuo(Price1, Price2)
FROM @T;
-- Returns 10

SELECT Category, dbo.MoneyMaxDuo(Price1, Price2)
FROM @T
GROUP BY Category;
-- Returns
-- A     7
-- B     10
```

Note that both parameters are required, so if you only need to pass one, use NULL for the other parameter; and be sure that your CLR code accommodates NULL parameter values.

Large User-Defined Types

Like UDAs, user-defined types (UDTs) also support the 2 GB figure, in this case for storage. As done with UDAs, the attribute that defines the type of SQL Server object, in this case SqlUserDefinedType, sets the MaxByteSize named parameter to a value of -1 to specify up to 2 GB of data is supported, as shown here in this abridged code snippet:

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType
    (Format.UserDefined, MaxByteSize = -1)]
public class EncryptedString : INullable, IBinarySerialize
{
...
}
```

When defined as such, the UDT is conceptually like VARBINARY(MAX), and for down-level clients, it is converted to VARBINARY(MAX) or IMAGE as appropriate.

Null Support

Passing parameters with null values into CLR-based code meant that you always needed to do a check to see if the value was indeed null and, if so, take appropriate action, as shown here:

```
[Microsoft.SqlServer.Server.SqlFunction]
public static SqlInt32 SquareWithNull(SqlInt16 input)
{
    if (input.IsNull == true)
        return SqlInt32.Null;
    return input * input;
}
```

A call to this user-defined function in SQL Server may pass a null value into the method, so the code checks to see if the parameter has a null value. If true, it returns a null value. Otherwise, the method returns the square of the parameter value. In SQL Server 2008,

however, you can now use the .Net Nullable types and do away with the extra check for each incoming parameter, as shown here:

```
[Microsoft.SqlServer.Server.SqlFunction]
public static Nullable<Int32> SquareWithNull(Nullable<Int16> input)
{
    return input * input;
}
```

Order Awareness

Table-valued user-defined functions have another new feature that allows you to specify the order of the results of CLR-based table-valued user-defined functions (UDFs). If you know that your UDF always returns results in a particular order, then you can optimize the UDF to take advantage of that and not do the extra sort work when selecting the results in that order.

To demonstrate, I will use the following UDF, which is deployed to the AdventureWorksLT database using Visual Studio.

```
public partial class UserDefinedFunctions
{
  [SqlFunction (FillRowMethodName = "FillRow",TableDefinition = @"TheNumber int")]
  public static IEnumerable fnList()
  {
    return new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  }

  public static void FillRow(Object obj, out SqlInt32 TheNumber)
  {
    TheNumber = (Int32)obj;
  }
}
```

The database now has a function named fnList, which will return a simple table with a single column, and 10 rows, in order from 0 to 9.

Now because you already know the order of the returned data, you can optimize the call. I'll add another UDF using the same underlying CLR method, as follows.

```
USE AdventureWorksLT
GO

CREATE FUNCTION [dbo].[fnList2]()
```

```
RETURNS  TABLE (
    [TheNumber] [int] NULL
) WITH EXECUTE AS CALLER
ORDER ([TheNumber])
AS
EXTERNAL NAME [OrderTVF].[UserDefinedFunctions].[fnList]
GO
```

The ORDER clause in the UDF definition tells the optimizer that the UDF returns its data in this particular order. And if the request for the data from the UDF is ordered the same, it will not need to sort (although it will check the order as it streams the data out).

Examine the following two SELECT statements.

```
SELECT TOP 10 * FROM dbo.fnList() ORDER By TheNumber;
SELECT TOP 10 * FROM dbo.fnList2() ORDER By TheNumber;
```

The results of each are identical, as you probably expected, but their execution plans are different, as shown here.

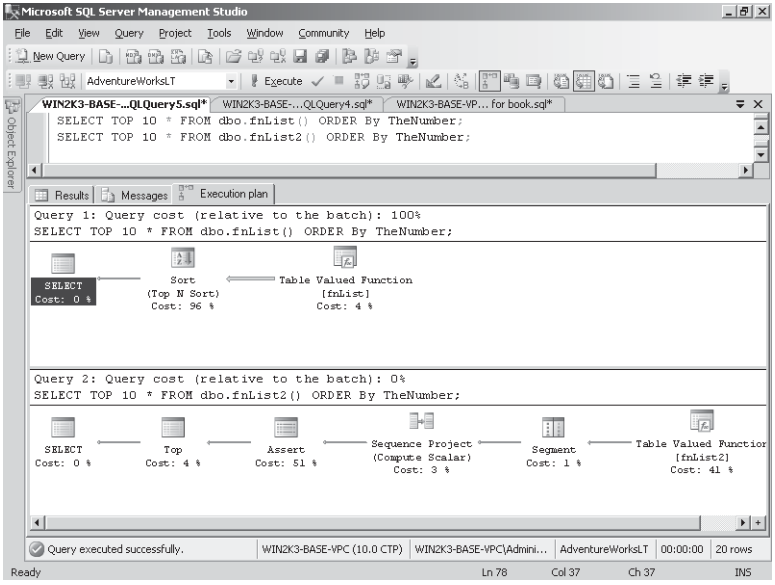


FIGURE 4-1 Typical and order optimized execution plans

The first execution plan shows the optimizer uses a sort operator to ensure the data is in order. The second execution plan seems more complicated, but because it is not using a sort

operator, it executes faster. On a more complicated example, I did some performance testing and found that the ORDER-optimized UDF ran faster, but how much faster varied based on the number of rows returned. As the number of rows returned decreased (by decreasing the TOP value), the difference in speed increased. So when I returned 100 out of 1000 rows, the speed difference was about three times faster for the ORDER-optimized UDF. But when I returned 10 of 1000 rows, the difference in speed was almost an order of magnitude.

System CLR Types

The CLR infrastructure now has system CLR types, which include the new HIERARCHYID data type, the new GEOMETRY and GEOGRAPHY spatial data types, and the Policy-Based Management features. The new system CLR data types are implemented in .NET yet do not require the CLR Enabled option to be turned on in order to use them.

SQL Server Management Studio Enhancements

Although not technically a programming topic, a number of new enhancements in SSMS directly relate to development work productivity.

Intellisense

Most .Net developers are already familiar with the awesome ability of Intellisense. There are even third-party products that incorporate Intellisense into products such as Visual Studio and SSMS. SQL developers have been asking for this latest feature for many years. And Microsoft has delivered.

By examining the options for Intellisense, you can get a better idea of what features it really offers. Figure 4-2 shows the Intellisense options (located under Text Editor, Transact-SQL, and then Advanced in the nodes) in the Options dialog box.

If Intellisense is enabled, then you can have it also auto list members and give parameter information (as shown in Figure 4-3). If you prefer only to invoke such features when you want, then you can turn these off and easily invoke using Ctrl+J to auto list members or Ctrl+Space to complete unique member names, or list members if not unique.

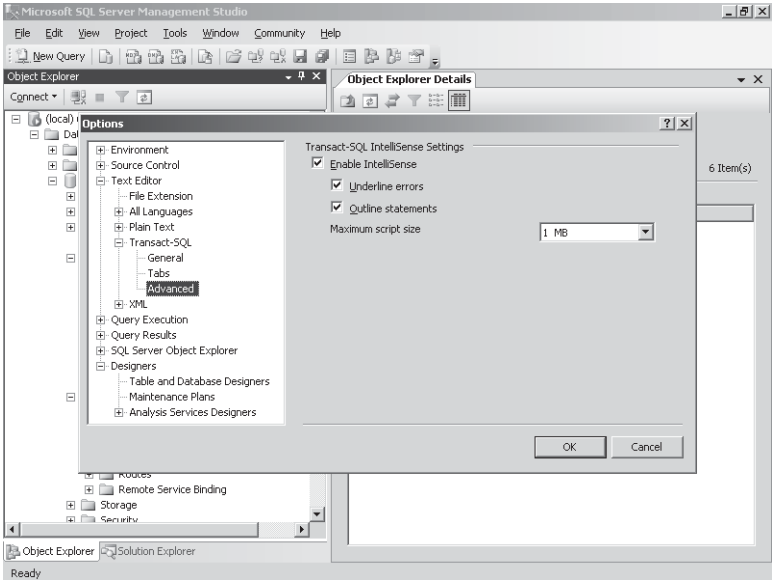


FIGURE 4-2 Intellisense settings

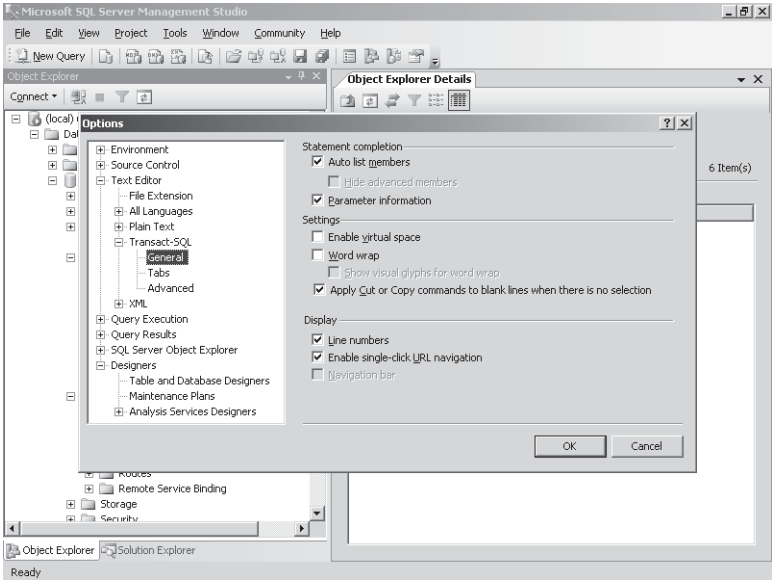


FIGURE 4-3 Statement Completion options

Once in effect, you can see the benefits right away. Figure 4-4 shows the auto complete kicking in after typing the period in "HumanResources." You can see a list of valid table and views from the HumanResources schema in the list.

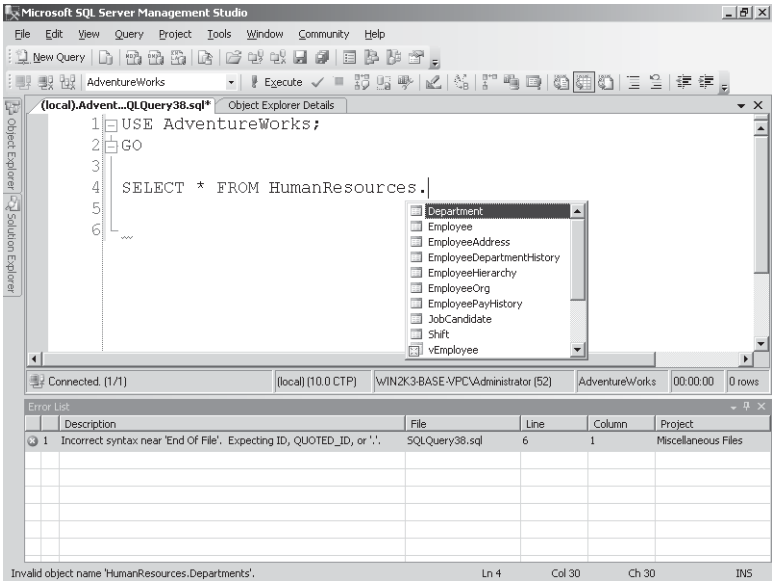


FIGURE 4-4 Intellisense in action

The list shows other objects, such as functions, columns, and so on, depending on the context in which it invokes or is invoked. But that's not all Intellisense can do....

Error List

Not only will the T-SQL editor now outline the statements, but it will also display a list of errors that it sees in the code. The new Error List window shows you two kinds of errors: syntax and reference. Double-clicking the error in the Error List will bring you to the code in question in the T-SQL editor. Sometimes a simple mistake shows up as multiple errors, as shown in Figure 4-5.

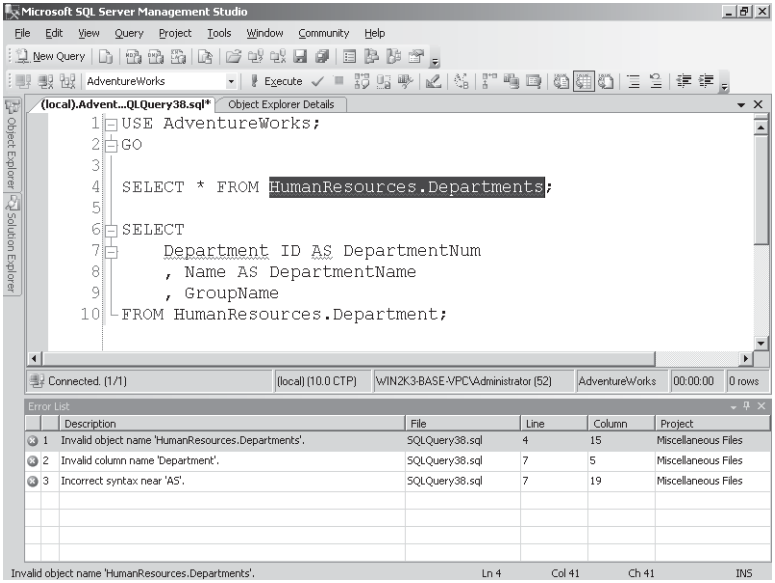


FIGURE 4-5 The Error List in SQL Server Management Studio

The first error is simply that the name of the table is Department and not Departments. But the typo Department ID (instead of DepartmentID) shows up as two errors. The first is a reference error, because Department is not a column in the table. The second is a remnant of the first: Because there is an extra "word" in the code, it thinks the code is really trying to select from the Department column with an alias of ID, at which point it expects a comma and instead it finds the word AS.

Service Broker Enhancements in SSMS

Like SQL Server 2005, you can view Service Broker objects using SSMS. Each database node in Object Explorer now has a Service Broker child node, which in turn has child nodes for each of the Service Broker object types, as shown in Figure 4-6.

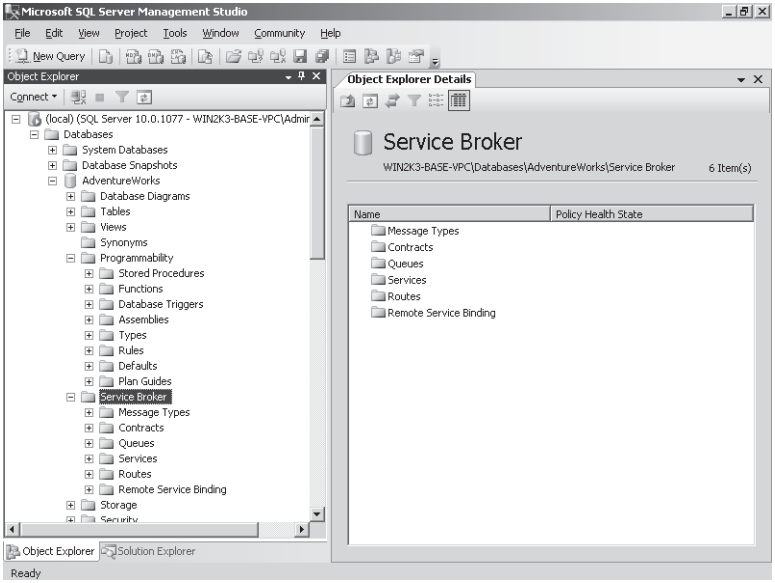


FIGURE 4-6 Service Broker in Object Explorer

However, new to SQL Server 2008, you can also right-click on a Service Broker object and take action or view its properties. For example, you can right-click Services and choose to create a new initiator or target service. This action will result in a new query window opening, with a T-SQL template for the selected object. Coding Service Broker just got easier. Also, from the context menu, you can also choose to view properties of a Service Broker object, as shown in Figure 4-7.

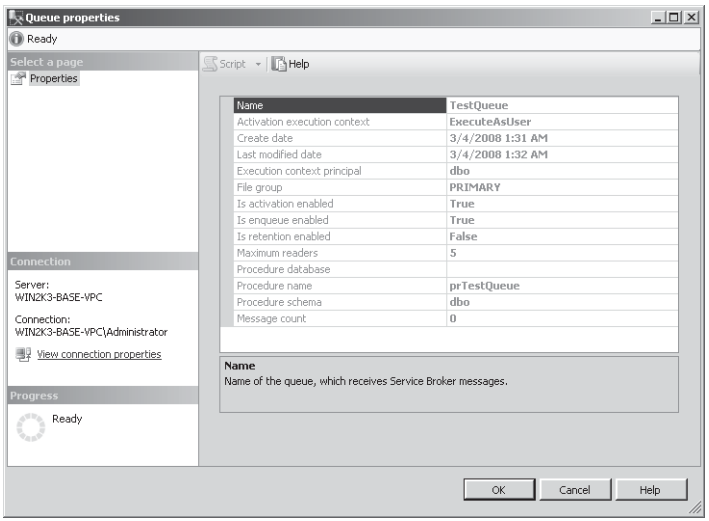


FIGURE 4-7 Viewing a Service Broker queue's properties

PowerShell

In case you didn't know, Windows PowerShell is a command-line shell and scripting language that is quite the hot topic nowadays. One of the other cool features that SQL Server 2008 includes is PowerShell support for SQL Server, and SSMS can be used to access PowerShell right from the Object Explorer. For example, you can navigate to AdventureWorksLT database and right-click on the Tables node. Choose Start PowerShell from the context menu, and you will see the SQL PowerShell (SQLPS), as shown in Figure 4-8.

FIGURE 4-8 SQL PowerShell

Notice how the path shown in the prompt is the same as the node hierarchy in Object Explorer in SSMS (*server name*, *instance name*, Databases, *database name*, Tables). From here, you can issue a simple cmdlet such as Get-ChildItem, and you will see these results.

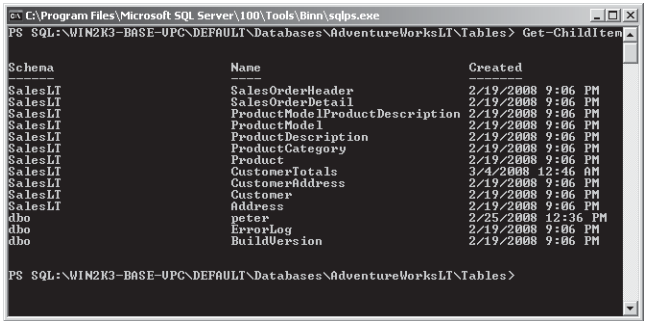


FIGURE 4-9 Listing tables in SQL PowerShell

Let's say I need to get rid of that table named Peter. To do so, I simply run the following cmdlet.

```
Remove-Item dbo.Peter
```

You can filter the list shown to limit to those tables in the SalesLT schema, as follows.

```
Get-ChildItem | where {$_.Schema -eq "SalesLT"}
```

You can also navigate using the Set-Location cmdlet (it works just like *CD* does in Command prompt). For example, to move up a level:

```
Set-Location ..
```

To move down to the Views node:

```
Set-Location Views
```

But that is only scratching the surface. You can use variables, iterate through properties, interact with the file system, and even script objects. SQL Server also includes snap-ins for PowerShell, such as Invoke-Sqlcmd, which works like Sqlcmd to execute T-SQL or XQuery script. You can also manage SQL Mail, SQL Agent, Service Broker, and even the new Policy-Based Management from PowerShell. The usefulness of PowerShell seems to have no bounds.

> **More Info** If you want to learn more about it, you can visit Microsoft's TechNet Web site, and you can also read the "SQL Server PowerShell Overview" topic in SQL Server Books Online.

Summary

There are some phenomenal new features available in T-SQL in SQL Server 2008. All of them are designed to make your programming experience better, to give you more flexibility when writing code in T-SQL, and to increase performance.

Chapter 5

Storage

Introduction

SQL Server 2008 has added two new storage-related features that can dramatically reduce the storage requirements of sparsely populated tables and create targeted indexes using criteria that can not only reduce the index storage requirements but increase performance for both read and modify operations.

Sparse Columns

Not too long ago, I was working with a client who was implementing software using SQL Server 2005 Express. The objective was to deliver sales data to various clients who were in turn doing some analysis of the data. There was one problem, however, in that some of the data files to be delivered topped out at more than 4 gigabytes (GBs), which is the maximum size allowed for SQL Server 2005 Express Edition.

The problem was that there were a lot of null values in the data that were taking up a lot of room. Changing the relational model to a key-value pair solution shrank the data files quite a bit but slowed the querying of the data. A colleague solved the problem by implementing a user-defined data type and some well-written C#-based user-defined functions and stored procedures.

The solution to this problem would have been almost infinitely simpler, however, using sparse columns.

What Is a Sparse Column?

A sparse column is meant to be used when the column will contain a majority of null values. It is optimized for null storage and actually uses no space for null values. There is a trade-off, however, in that non-null values require extra space for storage.

- Null values require no space when being stored.
- Non-null values require extra bytes of storage space.
  - Fixed-length and precision-dependent types require 4 extra bytes.
  - Variable-length types require 2 extra bytes.

If the column contains few null values, then using a sparse column is not beneficial. If, however, there were few non-null values in that column, then a sparse column could considerably reduce your storage requirements.

So, for example, let's say we have the following table.

```
CREATE TABLE [dbo].[Customer]
(
    [CustomerID] INT PRIMARY KEY,
    [Name] VARCHAR(100),
    [LocationCount] INT NULL
)
```

And let's say that there are 1,000 customers stored in that table, but only 200 of them have reported how many locations they have. Although the [LocationCount] value will be null for 800 of these rows, the column still requires 4,000 bytes of storage (1,000 x 4 bytes for INT).

If you change that column to a sparse column, however, those figures change.

```
CREATE TABLE [dbo].[Customer]
(
    [CustomerID] INT PRIMARY KEY,
    [Name] VARCHAR(100),
    [LocationCount] INT NULL SPARSE
)
```

For the [LocationCount] column to use less space, at least 50 percent of its values need to be null. In this case, 80 percent of the values are null. The remaining 200 non-null values now take up 1,600 bytes (200 x [4 + 4 bytes overhead]), and the 800 null values require no space, so the total amount of space used for that column drops from 4,000 bytes to 1,600 bytes—a savings of 60 percent. Extrapolate this to more columns and more rows and, all of a sudden, your database storage requirements drop significantly.

When to Use Sparse Columns

Sparse columns denormalize data. How you implement sparse columns will determine the how much positive impact it will have on performance. I ran some performance tests on two tables, shown here:

```
CREATE TABLE [dbo].[TableTest1]
(
    [TableID] INT NOT NULL,
    [CharCol] CHAR(20) NULL,
    [IntCol] INT NULL
)

CREATE TABLE [dbo].[TableTest2]
(
    [TableID] INT NOT NULL,
    [CharCol] CHAR(20) SPARSE NULL,
    [IntCol] INT SPARSE NULL
)
```

I tested the following scenarios:

- Always inserting a value into the sparse columns
- Never inserting a value into the sparse columns
- Inserting mostly non-null values into the sparse columns (in varying percentages)
- Inserting mostly null values into the sparse columns (in varying percentages)

As you may have surmised, when a lot of non-null data is inserted, inserting into the table without sparse columns performed better than inserting into the table with the sparse columns; the table without sparse columns also required less storage space. As the amount of null data increased, however, inserting into the table with sparse columns resulted in better performance and required less storage for the table with the sparse columns.

So when should you use a sparse column? You must look at the frequency of null in that column. If the column will contain more null values than non-null values, it is a candidate to be a sparse column. For example, a flattened table will often result in columns that contain many null values.

I recently worked on a project where an object-relational mapping (ORM) solution was being used, and the domain model contained several concrete classes that inherited from a single abstract class. This scenario also lends itself to using sparse columns.

To exemplify such a scenario, let's start with an abstract class named *Person* and four concrete subclasses named Student, Professor, StaffMember, and Benefactor, as shown in Figure 5-1.

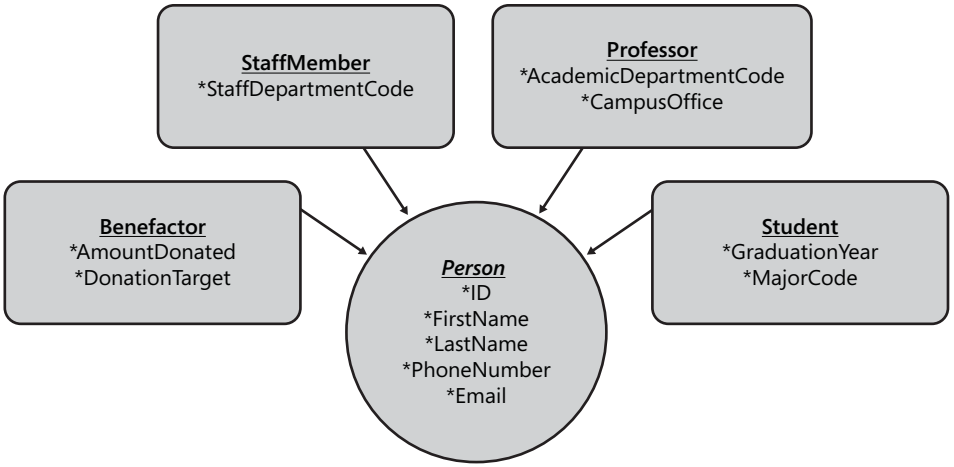


FIGURE 5-1 Diagram of an abstract class named *Person* and four concrete subclasses

There are several ways to implement the relational model for this solution. The first is to represent each class, abstract or concrete, as a table in the database and to have the concrete class tables each have a foreign key to the abstract class table, as shown in the following listing.

```
CREATE TABLE Person
(
    PersonID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
)
GO

CREATE TABLE Benefactor
    PersonID INT NOT NULL PRIMARY KEY
        FOREIGN KEY REFERENCES Person(PersonID)
    , AmountDonated MONEY NOT NULL
    , DonationTarget VARCHAR(100) NOT NULL
)
GO

CREATE TABLE StaffMember
(
    PersonID INT NOT NULL PRIMARY KEY
```

```
        FOREIGN KEY REFERENCES Person(PersonID)
    , StaffDepartmentCode CHAR(5) NOT NULL
)
GO

CREATE TABLE Professor
(
    PersonID INT NOT NULL PRIMARY KEY
        FOREIGN KEY REFERENCES Person(PersonID)
    , AcademicDepartmentCode CHAR(5) NOT NULL
    , CampusOffice VARCHAR(25) NOT NULL)
GO

CREATE TABLE Student
(
    PersonID INT NOT NULL PRIMARY KEY
        FOREIGN KEY REFERENCES Person(PersonID)
    , GraduationYear SMALLINT NOT NULL
    , MajorCode CHAR(4) NOT NULL
)
GO
```

The preceding implementation is the most normalized. Others, however, may prefer to represent each concrete class as a table and to implement the abstract properties multiple times (once for each concrete class), as shown in the following listing.

```
CREATE TABLE Benefactor
    BenefactorID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , AmountDonated MONEY NOT NULL
    , DonationTarget VARCHAR(100) NOT NULL
)
GO

CREATE TABLE StaffMember
(
    StaffMemberID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , StaffDepartmentCode CHAR(5) NOT NULL
)
```

```
GO

CREATE TABLE Professor
(
    ProfessorID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , AcademicDepartmentCode CHAR(5) NOT NULL
    , CampusOffice VARCHAR(25) NOT NULL
)
GO

CREATE TABLE Student
(
    StudentID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , GraduationYear SMALLINT NOT NULL
    , MajorCode CHAR(4) NOT NULL
)
GO
```

Still others may prefer to represent all concrete classes using a single table that has all fields from all the concrete classes. This particular implementation lends itself to using sparse columns, as shown in the following listing.

```
CREATE TABLE Person
(
    PersonID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , PersonType TINYINT NOT NULL
    , AmountDonated MONEY NULL
    , DonationTarget VARCHAR(100) NULL
    , StaffDepartmentCode CHAR(5) NULL
    , AcademicDepartmentCode CHAR(5) NULL
    , CampusOffice VARCHAR(25) NULL
    , GraduationYear SMALLINT NULL
    , MajorCode CHAR(4) NULL
)
```

Tip You will want to ensure that you have the required data for the particular type of person—perhaps something like this CHECK CONSTRAINT.

```
ALTER TABLE Person
ADD CONSTRAINT chkPersonType CHECK
((PersonType = 1 AND AmountDonated IS NOT NULL AND DonationTarget IS NOT NULL)
  OR (PersonType = 2 AND StaffDepartmentCode IS NOT NULL)
  OR (PersonType = 3 AND AcademicDepartmentCode IS NOT NULL
      AND CampusOffice IS NOT NULL)
  OR (PersonType = 4 AND GraduationYear IS NOT NULL AND MajorCode IS NOT NULL)
)
```

You could also implement several different check constraints or even a trigger to ensure data integrity.

Each of the implementations has certain advantages and disadvantages in regards to design, performance, normal form, and so on—all of which can be debated for or against. Of the three designs, however, the latter design lends itself to use sparse columns, as shown here.

```
CREATE TABLE Person
(
    PersonID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , PersonType TINYINT NOT NULL
    , AmountDonated MONEY SPARSE NULL
    , DonationTarget VARCHAR(100) SPARSE NULL
    , StaffDepartmentCode CHAR(5) SPARSE NULL
    , AcademicDepartmentCode CHAR(5) SPARSE NULL
    , CampusOffice VARCHAR(25) SPARSE NULL
    , GraduationYear SMALLINT SPARSE NULL
    , MajorCode CHAR(4) SPARSE NULL
)
```

You also could have altered each of the columns using an ALTER TABLE statement. For example, to make the DonationAmount column be sparse, you could execute the following.

```
ALTER TABLE Person
ALTER COLUMN DonationAmount ADD SPARSE
```

You can also turn off the sparse feature for a column as follows (again, using DonationAmount as the example):

```
ALTER TABLE Person
ALTER COLUMN DonationAmount DROP SPARSE
```

Note Altering a column to be sparse or non-sparse requires that the storage layer restore the data in the sparse/non-sparse format. Therefore it is advisable to either set the sparse attribute of a column when creating the table or only to alter the sparse attribute of a column on a table with little or no data.

Regardless of the method you use to create the sparse columns in the table, the sparse column solution has a distinct advantage over the other two designs: It will not only require less space to store the data but will have better overall performance than the other two proposed models. And it also allows you to use the column sets feature (discussed later in this section).

Sparse Column Rules and Regulations

There are a number of details that you should keep in mind when designing a table with sparse columns. The most obvious rule is that the column must be nullable because the column will contain mostly null values. You also cannot assign a default or bind any rules to a sparse column, nor can you use IDENTITY or ROWGUIDCOL on said column. If the column has any of these features or attributes, it cannot be altered to be a sparse column.

Sparse columns support all data types except *GEOGRAPHY*, *GEOMETRY*, *TEXT*, *NTEXT*, *IMAGE*, *TIMESTAMP*, user-defined data types, and *VARBINARY(MAX)* that are *FILESTREAM*. Computed columns cannot be marked as sparse, although you can use a sparse column in the calculation of a computed column.

Sparse columns cannot be directly part of a clustered index or unique primary key index, nor can they be used as a partition key of a clustered index or heap. They can, however, be used as the partition key of a nonclustered index.

A table normally has a row size limit of 8,060 bytes. When using sparse columns in a table, this row size limit is reduced to 8,018 bytes. You can have as many as 30,000 columns in a table with sparse columns (yes, you read that correctly), as long as each row is less than the 8,018 byte size limit. You can also have 30,000 statistics and 1,000 indexes defined in a table with sparse columns.

Note Variable length data (varchar, varbinary, and so on) relaxes the row size restriction, because variable length data can be stored off-row. This relaxation of the row size restriction applies to tables both without and with sparse columns. See the "Row-Overflow Data Exceeding 8 KB" topic in SQL Server Books Online for more details.

Here are a few other items of note:

- A table with sparse columns cannot be compressed.
- Table types cannot use sparse columns.
- Sparse columns can be used with transactional replication, change tracking, and changed data capture but cannot be used with merge replication.

Column Sets

When using the Person table with sparse columns, what if you wanted to return only the fields for a particular person type, such as a student? One option would be to use a view, as shown here.

```
CREATE VIEW vwStudent
AS
    SELECT
        PersonID
        , FirstName
        , LastName
        , PhoneNumber
        , Email
        , GraduationYear
        , MajorCode
    FROM Person
    WHERE PersonType = 4
GO
```

The other option, however, is to use a column set that will return the non-null sparse columns composed as XML. To do this, we need to add another column to the table, as shown here.

```
CREATE TABLE Person
(
    PersonID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
```

```
    , Email VARCHAR(128) NULL
    , PersonType TINYINT NOT NULL
    , AmountDonated MONEY SPARSE NULL
    , DonationTarget VARCHAR(100) SPARSE NULL
    , StaffDepartmentCode CHAR(5) SPARSE NULL
    , AcademicDepartmentCode CHAR(5) SPARSE NULL
    , CampusOffice VARCHAR(25) SPARSE NULL
    , GraduationYear SMALLINT SPARSE NULL
    , MajorCode CHAR(4) SPARSE NULL
    , Details XML COLUMN_SET FOR ALL_SPARSE_COLUMNS
)
```

Note You cannot add an XML column set column to a table that already has sparse columns.

Note A column set column is similar to a computed column and does not use any physical storage space on the disk.

There can be only one column set column per table, and the syntax will always be the following:

```
<column_name> XML COLUMN_SET FOR ALL_SPARSE_COLUMNS
```

To insert a row of data, you could then execute the following statement, for example.

```
INSERT INTO Person
    (PersonID, FirstName, LastName, PhoneNumber, Email, PersonType,
     GraduationYear, MajorCode)
VALUES (1, 'Jane', 'Doe', '555-3434', 'jane.doe@college.edu', 4, 2008, 'CMSC')
```

And to see the effects of this, you can then issue the following statement.

```
SELECT * FROM Person
```

What you get back, however, might be a little surprising.

PersonID	First Name	Last Name	Phone Number	Email	Person Type	Details
1	Jane	Doe	555-3434	jane.doe@college.edu	4	<GraduationYear> 2008 </GraduationYear> <MajorCode> CMSC </MajorCode>

You will notice that none of the sparse columns are returned, and, instead, all non-null sparse column values are composed as XML and returned in the column set. This doesn't mean you can no longer return sparse columns, but, rather, you simply need to explicitly list the sparse columns in the SELECT statement in order to return them.

Note Column sets provide yet another reason not to use SELECT * FROM in production code.

Another very cool thing is that the column set can be modified. For example, examine the following code.

```
INSERT INTO Person
    (PersonID, FirstName, LastName, PhoneNumber, Email, PersonType, Details)
VALUES (2, 'John', 'Doe', '555-1212', 'john.doe@college.edu', 4,
    '<GraduationYear>2009</GraduationYear><MajorCode>ECON</MajorCode>')
```

You are not seeing a typographic error. You can pass the sparse column values in as XML in the INSERT statement, and it will insert the values in the sparse columns. I've implemented a similar feature using views instead of triggers, and I can tell you firsthand, column sets are so very much better. You can also use the column set to update the sparse column values, as shown here.

```
UPDATE Person
SET Details = '<GraduationYear>2009</GraduationYear><MajorCode>LING</MajorCode>'
WHERE PersonID = 2
```

You should note that all sparse columns are modified when using the column set to insert or update, so the statement above is not simply updating the GraduationYear and MajorCode

columns but also updating all the other sparse columns to null. So if you execute the following:

```
UPDATE Person
SET Details = '<MajorCode>EENG</MajorCode>'
WHERE PersonID = 2
```

It would set MajorCode to EENG, and all other sparse columns, including the GraduationYear, to null.

Note If the CHECK constraint shown earlier in the section were implemented, this statement would fail the check and throw an exception.

You've seen that when the table has a column set, the default set of columns (SELECT * operations) does not contain the sparse columns. This also holds true when inserting data into the table. Examine the following code.

```
INSERT INTO Person
VALUES (3, 'Jack', 'Doe', '555-5656', 'jack.doe@college.edu', 4,
    '<GraduationYear>2010</GraduationYear><MajorCode>LING</MajorCode>')
```

Because the Person table has a column set, its default columns do not include the sparse columns; it is semantically equivalent to listing the non-sparse columns. That being said, I recommend always being explicit in your code.

Note You cannot modify both the column set and the sparse columns in a single INSERT or UPDATE statement. For example, the following is not allowed.

```
UPDATE Person
SET Details = '<MajorCode>EENG</MajorCode>'
    , GraduationYear = 2009
WHERE PersonID = 2
```

Executing this query would give this result:

```
Msg 360, Level 16, State 1, Line 1
The target column list of an INSERT, UPDATE, or MERGE statement cannot contain both
a sparse column and the column set that contains the sparse column. Rewrite the
statement to include either the sparse column or the column set, but not both.
```

And this restriction applies to all sparse columns, regardless of their use in the individual rows.

Filtered Indexes

In addition to sparse columns and column sets, two more related technological advances make their debut in SQL Server 2008: filtered indexes and filtered statistics.

Filtered Index

In simplest terms, a filtered index is an index with criteria; the CREATE INDEX statement now has an optional WHERE clause, which is used to specify the criteria. This can be used to refine existing indexes that favor certain subsets of data, or it can be used in conjunction with the new sparse column technology to further optimize sparsely populated tables.

For example, an index that contains both the PersonType and MajorCode columns would eventually get down to a leaf level containing a combination of PersonType and MajorCode, which point to the data, as shown in Figure 5-2.

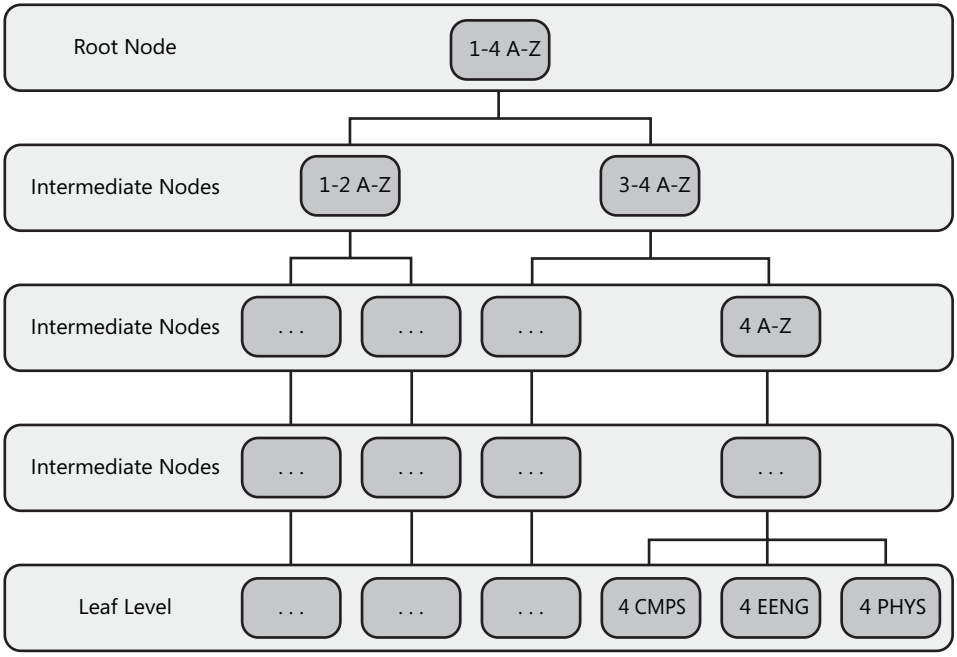


FIGURE 5-2 Index that contains both the PersonType and MajorCode columns

When using a filtered index, if we select to index only those rows that have PersonType = 4, rows with other values of PersonType would be ignored, regardless of whether they contained a MajorCode. The result: a smaller index targeting specific rows of data, as shown in Figure 5-3.

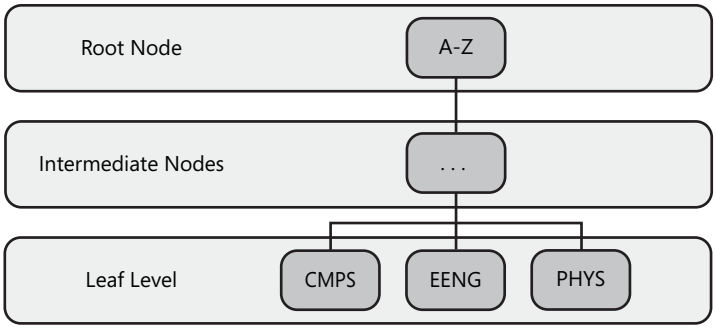


FIGURE 5-3 Index that targets specific rows of data

Using a Filtered Index

I will not attempt to explain the innards of the query optimizer in this section but rather provide a few guidelines as to when you might expect the optimizer to take advantage of a filtered index.

The probability of a filtered index being used is higher when the criteria in WHERE clause of the select statement matches the criteria of the WHERE clause of the filtered index. Let's re-visit the Person table from the sparse columns section earlier in this chapter.

```
CREATE TABLE Person
(
    PersonID INT NOT NULL PRIMARY KEY
    , FirstName VARCHAR(20) NOT NULL
    , LastName VARCHAR(30) NOT NULL
    , PhoneNumber VARCHAR(15) NOT NULL
    , Email VARCHAR(128) NULL
    , PersonType TINYINT NOT NULL
    , AmountDonated MONEY SPARSE NULL
    , DonationTarget VARCHAR(100) SPARSE NULL
    , StaffDepartmentCode CHAR(5) SPARSE NULL
    , AcademicDepartmentCode CHAR(5) SPARSE NULL
    , CampusOffice VARCHAR(25) SPARSE NULL
    , GraduationYear SMALLINT SPARSE NULL
    , MajorCode CHAR(4) SPARSE NULL
    , Details XML COLUMN_SET FOR ALL_SPARSE_COLUMNS
)
GO

CREATE NONCLUSTERED INDEX fiPersonStudentMajorCode
    ON Person (MajorCode)
WHERE PersonType = 4;
GO
```

```
CREATE VIEW vwStudent
AS
    SELECT
        PersonID
        , FirstName
        , LastName
        , PhoneNumber
        , Email
        , GraduationYear
        , MajorCode
    FROM Person
    WHERE PersonType = 4
GO
```

As mentioned earlier in the chapter, the sparsely populated column MajorCode will only have data for a person that is a type of student (when PersonType equals 4). If the index were created without the WHERE clause criteria, then a majority of the MajorCode field data would be null. By filtering for students, the index will now contain only non-null data, and finding students by major will be much more efficient. Examine the following SELECT statement:

```
SELECT
    PersonID
    , FirstName
    , LastName
    , PhoneNumber
    , Email
FROM vwStudent
WHERE MajorCode = 'EENG'
```

This statement is semantically equivalent to the following statement.

```
SELECT
    PersonID
    , FirstName
    , LastName
    , PhoneNumber
    , Email
FROM Person
WHERE PersonType = 4
 AND  MajorCode = 'EENG'
```

And this query takes advantage of the filtered index because the WHERE clause uses the same criteria as the filtered index's WHERE clause and because it additionally contains criteria

on the column defined in the filtered index. You might be inclined to think that the following query will also use the filtered index.

```
SELECT
    PersonID
    , FirstName
    , LastName
    , PhoneNumber
    , Email
FROM Person
WHERE PersonType = 4
```

In this case, the use of the filtered index depends on the cardinality of the data. Because there is likely going to be more rows in the Person table with PersonType = 4 (students make up a majority of the population at a university), it is actually more efficient to scan the clustered index than to use the filtered index to retrieve the data. If, however, there were few rows having PersonType = 4, then the filtered index would be used. This is similar to the way standard indexes work.

Now let's assume that very few people have an email address in the data (most are null). An appropriate filtered index for such a scenario might look like the following index:

```
CREATE NONCLUSTERED INDEX fiPersonWithEmailByName
    ON Person (FirstName, LastName)
WHERE Email IS NOT NULL;
```

And then execute the following query.

```
SELECT
    FirstName
    , LastName
    , Email
FROM Person
WHERE Email IS NOT NULL
```

You would expect the optimizer to use the fiPersonWithEmailByName index because it covers the query, but, in fact, it would still do a clustered index scan. Although the filtered index only contains records where the Email is not null, is doesn't contain the email itself. In this scenario, we also need the field from the WHERE clause of the filtered index to consider the query covered, as shown in this filtered index definition.

```
CREATE NONCLUSTERED INDEX fiPersonWithEmailByName
    ON Person (FirstName, LastName) INCLUDE (Email)
WHERE Email IS NOT NULL;
```

Filtered indexes also have less of a performance hit when modifying data. For example, assume you open registration on your Web site for students to register themselves for the coming school year (although most do not supply an email). You insert thousands of records in a very short amount of time, but only those with an email address will be affected by the presence of the fiPersonWithEmailByName index since the index only contains data for rows with a non-null email address.

So when should you use filtered indexes? The ratio of number of rows affected by the filtered index to the number of rows in the table should be small. As that ratio grows, you will be best served by a standard index. As that ratio gets smaller, you will best be served by a filtered index. So if more than 50 percent of the rows in the Person table had a non-null email, then the fiPersonWithEmailByName index could be more costly to maintain than a standard index and would be a poor choice.

Parameterized queries can also behave differently than expected. For example, let's say the following filtered index has been created (with PersonType included in order to allow it to cover the subsequent query).

```
CREATE NONCLUSTERED INDEX fiPersonStaffName
    ON Person (LastName, FirstName) INCLUDE (PersonType)
WHERE PersonType IN (2, 3);
```

And then you execute the following script.

```
DECLARE @personType TINYINT = 2

SELECT
    FirstName
    , LastName
FROM Person
WHERE PersonType = @personType
 AND LastName = 'Adams'
```

You would again be inclined to think that the filtered index would be used, but because the value of @staff is not known at compile time, the optimizer does a clustered index scan

instead. For this scenario, if you know that @personType will be 2 or 3, then you can add the expression from the WHERE clause of the filtered index to the query, as follows.

```
DECLARE @personType TINYINT = 2

SELECT
    FirstName
    , LastName
FROM Person
WHERE PersonType IN (2, 3)
 AND PersonType = @personType
 AND LastName = 'Adams'
```

The optimizer then uses the filtered index.

More Info See the topic "Filtered Index Design Guidelines" in SQL Server Books Online for more details.

Filtered Statistics

Where filtered indexes allow you to create optimized indexes for predefined subsets of data, filtered statistics do the same for statistics. There are several scenarios where filtered statistics can help query plan quality. One of those scenarios occurs when running queries that use filtered indexes. Fortunately, filtered statistics are automatically created for filtered indexes.

Another scenario is when you have data in a nonindexed column that correlates to data in another column. For example, if most staff and faculty have an email address, but most students and benefactors do not have an email address, the following statistics can help improve the query plan of the SELECT statement that follows it.

```
CREATE STATISTICS stsStaffEmail
ON Person (Email)
WHERE PersonType IN (2, 3);
GO

SELECT
    FirstName
    , LastName
    , Email
FROM Person
WHERE PersonType IN (2, 3)
 AND Email LIKE 'M%'
```

Summary

Sparse columns can reduce storage requirements for the table and increase performance by allowing more data to be in cache, thus reducing disk input/output (I/O). Filtered indexes can also reduce storage requirements for the index, also bettering the chance that the data will be in cache, thus reducing the amount of work the engine must do to find the data via the index.

Chapter 6

Enhancements for High Availability

With every version of SQL Server, the options available to database administrators (and their organizations) for ensuring high availability of corporate data have been improving. SQL Server 2005 made a particularly notable stride forward by introducing database mirroring. In this chapter, we will look at how both database mirroring and clustering have been improved yet again in SQL Server 2008 in terms of performance, reliability, and manageability.

Database Mirroring Enhancements in SQL Server 2008

Database mirroring was introduced in SQL Server 2005 and fully supported since Service Pack 1. It has proved to be a very popular, high-availability mechanism for several key reasons:

- It does not require special hardware and operates on essentially a shared nothing arrangement. The systems are only joined by the network cable between them.
- It was included in the Standard Edition of the product.
- It is easy to set up and manage.
- Applications can be largely unaware that mirroring is being used.

Two basic styles of mirroring were included: synchronous and asynchronous. In both cases, the primary server would wait until its transaction log entries were written before proceeding.

With synchronous mirroring, the primary server also waited for the secondary server to make the same writes. This provided a high degree of security at a small performance cost. Where the performance cost of synchronous mirroring was too great (usually because of significant network latency), asynchronous mirroring could be used. This meant that the primary server did not wait for the write on the secondary server. This provided a higher performing arrangement but with a greater potential for data loss. Even though this data loss was usually minor, some systems cannot tolerate any data loss and need to work synchronously.

SQL Server 2008 improves the database mirroring story, in terms of performance, reliability, and manageability.

Automatic Page Repair

If one server in a mirror pair detects a corrupted page in the mirrored database, SQL Server 2008 will now attempt to recover a valid copy of the page from the other server. If it is successful, the corrupt page will be replaced by the valid copy, and the error will have been corrected.

The recovery action is initiated for the following SQL Server errors:

- **Error 823** An operating system error, such as a cyclic redundancy check (CRC) error, occurred while reading the page.
- **Error 824** A logical data error, such as a torn write or the detection of an invalid page checksum, occurred while interpreting a page that has been read.
- **Error 829** An attempt was made to read a page that has been flagged as having a previous restore attempt still pending.

Most pages in a database can be recovered. There are a few exceptions, however, such as file header pages, database boot pages, and any of the allocation pages such as Global Allocation Map (GAM), Shared Global Allocation Map (SGAM), and Page Free Space (PFS) pages.

To provide a typical example, I have executed a query against an intentionally damaged copy of *AdventureWorks*. Note in Figure 6-1 that an error has occurred while reading some rows.

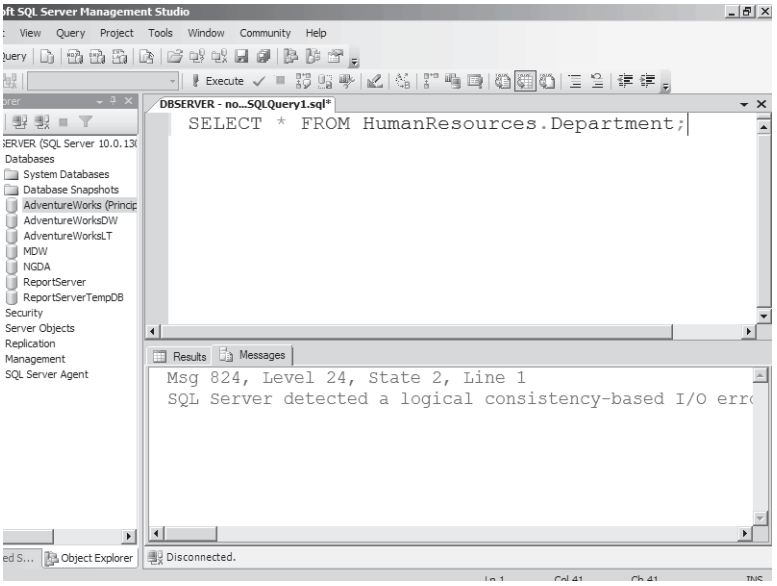


FIGURE 6-1 SQL Server Error 824 during page read

After waiting a short while, note in Figure 6-2 that the error has now been corrected automatically, and the command is now successful when executed again.

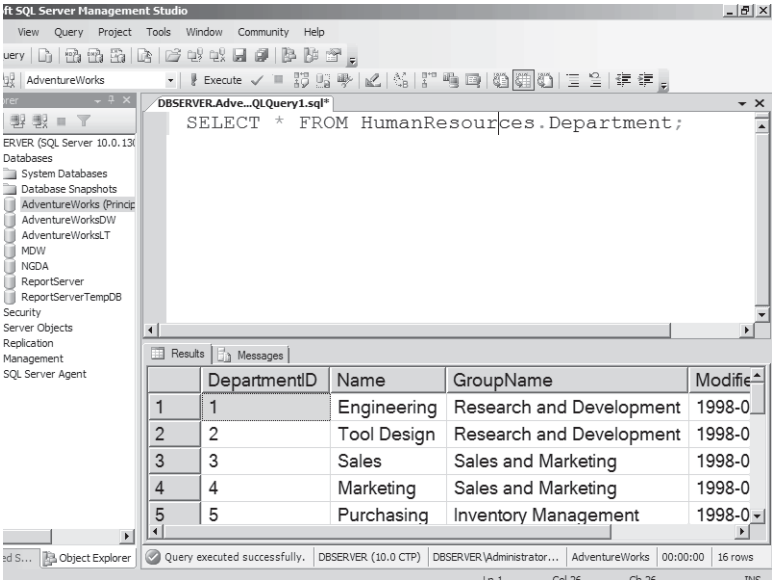


FIGURE 6-2 Read now successful after automatic recovery

Suspect Pages Table

When attempting to recover the corrupted page from the mirror partner, servers also log details of the problems with the page in the suspect_pages table in the *msdb* database. This table holds a row for each corrupt page, containing the database id, the file id, and the page id. In addition, it holds details of which type of event has occurred, how many times it has occurred, and when it last occurred.

It is possible to work out the status of a corrupt page recover by reading the details in this table for the page in question. The event type will be 1, 2, or 3 for a suspect page and 4 or 5 for a page that has been restored successfully. Other event types are used by database consistency checks (DBCCs).

Figure 6-3 shows the entry in this table that occurred from executing the command against the damaged database in the last section.

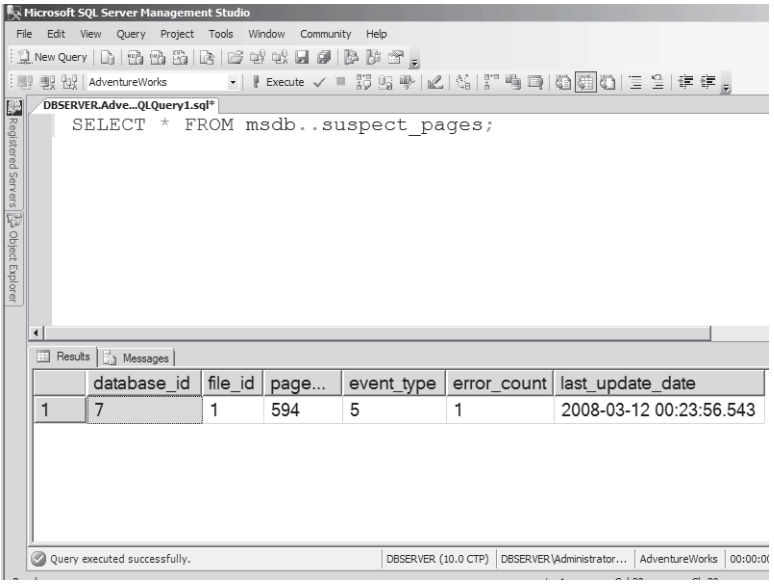


FIGURE 6-3 Suspect page table entry

A Note on Hardware Reliability

While having the ability to automatically recover damaged database pages can help improve the uptime and availability of a database server, it is really important to note that the occurrence of errors like 823 and 824 typically indicates a problem in the underlying hardware and operating system infrastructure that requires attention.

In a correctly operating system, pages don't mysteriously just become corrupt. The most common causes of these problems are

- A disk subsystem error (this might indicate faulty disk drives, controllers, storage area network [SAN] controllers, SAN cache memory corruption)
- Main system memory corruption (this might indicate a faulty memory subsystem or some really poorly behaved and highly privileged code)

While monitoring SQL Server, if suspect page table entries are found, further investigation of the underlying system is almost always warranted.

Log Performance Enhancements

A number of enhancements have also been made to the performance of the database mirroring log stream. The most significant is that the log stream is now compressed. With any form of compression, there is always a trade-off against CPU load. Very few SQL Server

systems are CPU bound unless they have a basic problem like constant procedure recompilations. Most systems have a great deal of free CPU time available and are instead disk-bound.

Log stream compression reduces the amount of data that needs to be sent across the network connection and thus reduces the latency involved. As we discussed earlier, this is a key performance issue for database mirroring.

SQL Server 2008 also processes the incoming log stream more efficiently. It issues more asynchronous input/output (I/O) requests for incoming log blocks thus better utilizing the sequential write capability of the disk subsystem by combining more data into a single send.

The performance of the undo phase during failover has also been improved for situations where the primary server is still available online. Rather than waiting until it needs them, the secondary server will provide hints to the primary server regarding the pages it will likely need later. This allows the primary server to prefetch them so they are already available in the send buffer.

Transparent Client Redirection

In conjunction with database mirroring being introduced in SQL Server 2005, a new connection string property was also added to ADO.NET version 2. A typical connection string would look like the following:

```
"Server=PROD;Database=Prosperity;Trusted_Connection=true;Failover Partner=PRODMIRROR;C
onnect Timeout=40"
```

The Failover Partner entry was used to indicate where the secondary server was located. The use of this parameter was often misunderstood. When a connection to the primary server was first made, ADO.NET queried the primary server to ask if a mirror was being used. If so, ADO.NET then cached details of where the secondary server was located. The Failover Partner connection string entry was only needed when it was not possible to connect to the primary server at initial connection time; otherwise, the cached value from the primary server was used instead.

The cached value was not persisted between application starts and shutdowns. Many applications could not easily be used with database mirroring if their connection string details could not easily be modified. Ideally, the application would be unaware that the database was being mirrored.

With SQL Server 2008, the client library will now persist the cached value received from the primary server by storing this value in the registry. When a connection attempt is made and the primary server is unavailable, an attempt will be made to connect to the location persisted in the registry even if there is no Failover Partner property in the connection string.

This means that applications can make use of database mirroring quite transparently, without modification, as long as they can make an initial connection to the primary server.

SQL Server Clustering Enhancements

Clustering has been the premier high-availability option for SQL Server clients for a long time. In this area, SQL Server 2008 provides enhancements to both management and reliability.

Windows Server 2008 Clustering Enhancements

SQL Server 2008 leverages a number of enhancements that have been made to Microsoft Clustering Services in Windows Server 2008.

New Quorum Model

Windows Server 2008 clustering incorporates a new quorum model. The aim of the change is to remove single points of failure. The original design assumed that the disk storage for the quorum disk was always available and could be used to resolve issues when nodes lost communication with each other. Now, by default, each node gets a vote, as does the witness disk (which used to be referred to as the quorum disk). So, if the witness disk is lost but the nodes stay up, the cluster can stay available. It also provides for a majority-based cluster management scheme where you can fully configure which nodes get a vote.

The witness disk can now be a dedicated logical unit number (LUN), a small disk (at least 512MB) formatted as NTFS and does not require an allocated drive letter. It can even be a share on a file server that is not part of the cluster. And the file server can provide a witness disk for more than one cluster by providing multiple shares.

Up to 16 Nodes per Cluster

Windows Server 2008 clustering now supports up to 64 nodes per cluster. While, in theory, SQL Server might be able to work with all these cluster nodes, SQL Server is being tested with up to 16 nodes per cluster. For this reason, the supported configuration of SQL Server 2008 will be up to 16 nodes per cluster.

Subnets

While nodes still need to be connected to the same network, editing the subnet mask during setup is supported. This allows connections from multiple networks. Geo-clusters are also supported via virtual local area network (VLAN) technology. This is an abstraction layer that

allows two IP addresses to appear as a single address to the software layers above. This needs to be set up after the initial cluster setup process has been completed. All IPs must be up for the service to come online. Also supported are replicated or mirrored SAN environments that provide shared storage across cluster nodes.

GUID Partition Table Disks

One of the few remaining relics of the original PC system architecture was the structure of the hard drives. The Master Boot Record (MBR) structure was based around concepts of cylinders, heads, and sectors. GUID Partition Table (GPT) disks use a new partition table structure based on Logical Block Addressing (LBA). For SQL Server, the key advantages are

- A single partition greater than 2 TB in size can be created.
- GPT disks are more reliable as the GPT header and partition table are written both at the beginning and the end of the drive, for redundancy.

The layout of the GPT disks is as shown in Figure 6-4.

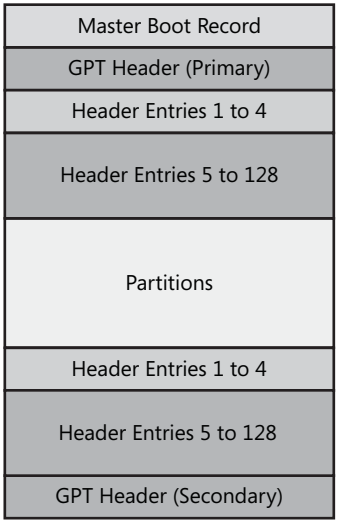


FIGURE 6-4 GPT disk layout

The MBR structure is still the starting place for the disk structure. It is then followed by the first copy of the GPT header, then the first four partition headers similar to that on an original MBR-based disk, followed by up to another 124 partition headers. After the partition data, there is another copy of all the partition headers and of the GPT header. This new structure provides for a more robust disk subsystem with built-in redundancy on the header entries, which are the most critical part of the disk.

Dynamic Host Configuration Protocol and IPv6 Support

While Network Address Traversal (NAT) technologies reduced the need for a rapid replacement of IP version 4, which is the current basis of the Internet, IP version 6, which will replace it, is slowly being deployed. Windows Server has provided an IPv6 communications stack for some time, and Windows Vista has detailed support for it as well. SQL Server 2008 now supports IPv6 for communication in clustered environments.

It also supports IP address assignments via Dynamic Host Configuration Protocol (DHCP). Where this is used, IP address resources will be automatically generated using DHCP for dependencies on network name resources. This is performed on a per-interface basis, and IP addressable resources will need to use the same method (DHCP or static addressing) as the node they are associated with.

SQL Server Cluster Setup and Deployment Improvements

SQL cluster setup is built on an entirely new architecture. It no longer has a dependency on remote task scheduling. This removes the source of many potential setup issues and should greatly improve the reliability of cluster node installations.

Cluster setup is now much more enterprise friendly. Cluster nodes are no longer deployed from a single node. Node deployment can now be performed reliably using unattended installation and service packs, and patches can be applied via command-line options. Enterprise deployments are further enhanced by support for the installation of cluster-prepared instances, which can then have failover clusters created from them in a single operation. Command-line options are also available to support this.

For smaller deployments, a simple integrated cluster installation option is provided. This allows for the configuration of a single node failover cluster in a single setup session. Additional nodes are then easily added with minimal interaction because feature selection and product properties are sourced directly from the active node.

Error reporting and handling for setup is also improved through a clear separation of operating system errors from SQL Server setup rule checks.

Rolling Upgrades and Patches

In earlier versions, a central installer was used to install and service all the nodes in a cluster at the same time. This meant that downtime would be experienced during such upgrades. The new installer in SQL Server 2008 involves a more distributed setup process, where each node is managed independently. The entire cluster no longer needs to be down during

the upgrade process. This can significantly improve the availability of the overall system by avoiding such downtime.

Upgrades are now performed in a side-by-side fashion. This allows for much more reliable upgrades. SQL Server instances on a cluster can be upgraded one node at a time with the database upgrade being the only period when the instance is unavailable for client connections.

Cluster Validation Tool

The System Consistency Checker (SCC) tool was a great addition to SQL Server 2005. Prior to SQL Server 2005, it was very common for administrators to install SQL Server without having their underlying systems configured appropriately. They would then have issues when using SQL Server that were directly caused by not having the prerequisite setup prior to installation. The SCC ran as the first stage of setup and performed two major functions:

- It checked all prerequisite configurations.
- Where the lack of one of these configured items would still allow an installation to proceed with reduced functionality, the tool would warn the administrator about the likely outcome.

The Cluster Validation Tool (CVT) now provides this same style of checking for prerequisite cluster configuration and is part of Windows Server 2008 failover clustering. SQL Server 2008 leverages this new validation tool by checking if the underlying operating system cluster has been validated and that no issues have been found. SQL Server 2008 failover cluster setup requires a successful CVT result on the operating system cluster. This helps separate operating system cluster issues from SQL Server setup issues. The CVT runs a set of tests on a set of servers that intended to be used together as a cluster and can help find problems before the cluster goes into production mode. It is also possible to use this tool in a diagnostic fashion once the cluster is up and running; however, when used in this mode, it will skip disks that are allocated to a service or application.

As an example of the type of checking the CVT will do, it starts by making sure each server is running the same version of Windows Server and that each server is at the same service pack and hotfix level. The use of the CVT removes the need for the strict Hardware Compatibility List (HCL) compliance that was required in earlier versions.

Similar to the way the SCC can still allow a SQL Server installation to be carried out with reduced functionality, it is possible for some CVT tests to fail but for the cluster to still install and function. However, systems that don't fully pass the tests undertaken by the CVT would not be treated as supportable by Microsoft Product Support.

High-Availability-Related Dynamic Management Views Enhancements

Dynamic Management Views (DMVs) were introduced in SQL Server 2005. They provide a detailed view of what's going on inside SQL Server. Mostly, they are doing this from data in memory structures rather than from any persisted data. SQL Server 2008 adds a significant number of DMVs, some of which are related to high availability.

dm_db_mirroring_auto_page_repair

This new DMV returns a row for every automatic page repair attempt on any mirrored database on the server instance. It keeps up to 100 rows per database. When more than 100 attempts have been made, this DMV returns the most recent 100 entries.

The view returns similar information to that stored in the suspect_pages table in the *msdb* database, including the database_id, file_id, page_id, error_type, page_status, and modification_time. The error_type returns -1 for hardware (823) errors, 1 for most 824 errors, 2 for bad checksums, and 3 for torn pages. The page_status returns 2 for a queued request, 3 for a request sent to the partner, 4 when queued for automatic page repair, 5 when this succeeds, or 6 if the page is not repairable.

dm_db_mirroring_past_actions

With all the internal enhancements to the SQL Server mirroring system, tracking the internal state of each mirror partner has become more complex. This view provides insights into the state machine actions being performed by the mirroring system. For each mirroring session (represented by a mirroring_guid), it provides entries for each state machine action taken (state_machine_name, the action_type and name, the current_state, and action_sequence).

Summary

While database mirroring was a major new improvement in SQL Server 2005, this latest version of SQL Server makes it an even stronger offering. In particular, database mirroring is now more robust through automatic page repair, higher performing through log compression, and easier to configure through transparent client redirection. New dynamic management views also provide additional insights into the operation of SQL Server mirroring.

SQL Server 2005 greatly increased the high-availability options available to SQL Server administrators. SQL Server 2008 makes these options much more manageable, reliable, flexible, and higher performing.

SQL Server 2008 clustering takes advantage of key enhancements to Windows Server 2008 clustering and is now easier to set up through the new setup tooling and the Cluster Validation Tool and easier to manage through the node-by-node rolling upgrade and patching mechanisms.

Chapter 7

Business Intelligence Enhancements

In this chapter, you will look at

- SQL Server Integration Services
- SQL Server Reporting Services
- SQL Server Analysis Services

SQL Server has made dramatic advances in all of these areas. We'll take a detailed look at some of the main changes for each of these technologies, and at the end of each section, we'll list the other new features that are worthy of further investigation.

SQL Server Integration Services Enhancements

Enhancements to SQL Server 2008 Integration Services (SSIS) have appeared in the areas of Extract, Transform, and Load (ETL); Lookup; and Data Profiling. Each has been significantly improved for flexibility and ease of use.

Performing ETL

There are two new enhancements to T-SQL designed for use in this area. The first is that INSERT INTO can now consume the output from a data manipulation language (DML) statement, and the second is the introduction of MERGE. Both are really useful in their own right, but, when INSERT INTO is used in conjunction with MERGE, the two work synergistically and offer huge benefits.

MERGE entered the SQL standard in 2003 for the simple reason that people had noticed that ETL is often complicated and becomes even more so when we have to deal with slowly changing dimensions. In the past, we have had to write multiple INSERT, UPDATE, and DELETE statements to achieve the results we wanted; MERGE enables us to combine these into a single statement.

For illustration, I'll use a very simple PRODUCT table and another very simple table (PDELTA) that contains the changes to be made to PRODUCT:

PRODUCT

PID	Name	Price	IsCurrent
1	Nut	0.45	Y
2	Bolt	0.75	Y
3	Washer	0.1	Y
4	Split washer	0.15	Y

PDELTA

PID	Name	Price	IsCurrent
1	Nut	0.55	Y
5	Screw	0.95	Y

Slowly Changing Dimension, Type 1

In a Type 1 slowly changing dimension, you want any new items inserted into the target table, but you do not wish to keep any record of historic data. And if the data about an existing item has changed, you simply want to overwrite previous values with new values. You can now do this with one MERGE statement:

```
MERGE Product as TRG
USING PDelta AS SRC
ON (SRC.PID = TRG.PID)
WHEN TARGET NOT MATCHED THEN
INSERT VALUES (SRC.PID, SRC.Name, SRC.Price, 'Y')
WHEN MATCHED THEN
UPDATE SET TRG.Price = SRC.Price;
```

MERGE: The Overview

Okay, so that's MERGE in action, but what is it doing, and how does it work?

A MERGE statement joins the source table to the target table, and the results from that join form the set upon which the DML statements within the MERGE statement operate.

Three types of WHEN clauses are supported within a MERGE statement:

- **WHEN MATCHED** When the source and target rows match, an UPDATE or DELETE action is performed on the relevant row in the target.
- **WHEN SOURCE NOT MATCHED** An UPDATE or DELETE action is performed on the relevant row in the target when the row exists in the target but not in the source.

- **WHEN TARGET NOT MATCHED** An INSERT action is performed to insert a row into the target when the row exists in the source but not in the target.

One MERGE statement can contain multiple WHEN clauses.

Search conditions can be included for each WHEN clause to select the type of DML operation to be carried out on the row.

So, the statement above essentially says:

Join the two tables on PID.

If there is a row in the source but not the target, insert it (with a "Y" in the IsCurrent column).

If they match, overwrite the price in the PRODUCT table.

MERGE allows us to add an OUTPUT like this:

```
…
UPDATE SET TRG.Price = SRC.Price
OUTPUT $action, SRC.PID, SRC.Name, SRC.Price;
```

Here you are making use of a virtual column titled $action, which displays, for each row, the DML action that was performed upon it. You see this result:

$action	PID	Name	Price
UPDATE	1	Nut	0.55
INSERT	5	Screw	0.95

As a side issue, this information can be very useful as a debugging aid, but it also gives you exactly what you need when working with Type 2 slowly changing dimensions.

Slowly Changing Dimension, Type 2

In such a dimension, you wish to retain the existing rows that contain the old values while marking them no longer current by setting IsCurrent to N. You then insert a row containing the new value with IsCurrent set to Y. You also wish to insert any rows of entirely new data.

```
MERGE Product as TRG
USING PDelta AS SRC
ON (SRC.PID = TRG.PID AND TRG.IsCurrent = 'Y')
WHEN TARGET NOT MATCHED THEN
INSERT VALUES (SRC.PID, SRC.Name, SRC.Price, 'Y')
WHEN MATCHED THEN
UPDATE SET TRG.IsCurrent = 'N'
OUTPUT $action, SRC.PID, SRC.Name, SRC.Price;
```

This statement gets parts of the job done: it sets IsCurrent to N for the current Nut row in the target and inserts the row for Screw, but it hasn't inserted the new row for Nut. But that's okay because you find the data you need in the output:

$action	PID	Name	Price
UPDATE	1	Nut	0.55
INSERT	5	Screw	0.95

The rows that still need to be inserted into the target table are labeled as INSERT in the column $action.

Remembering that INSERT INTO can now consume output from a DML statement, all you have to do is wrap a SELECT statement around the MERGE and then allow the INSERT INTO to consume that output.

```
INSERT INTO Product( PID, Name, Price, IsCurrent)
SELECT PID, Name, Price, 'Y'
FROM
(
MERGE Product as TRG
USING PDelta AS SRC
ON (SRC.PID = TRG.PID AND TRG.IsCurrent = 'Y')
WHEN TARGET NOT MATCHED THEN
INSERT VALUES (SRC.PID, SRC.Name, SRC.Price, 'Y')
WHEN MATCHED THEN
UPDATE SET TRG.IsCurrent = 'N'
OUTPUT $action, SRC.PID, SRC.Name, SRC.Price
)
As Changes (action, PID, Name, Price)
WHERE action ='UPDATE';
```

This does the job and inserts those final rows.

This is a very simple illustration of the power of these enhancements: The range of possibilities for using MERGE and INSERT INTO is huge, not only for slowly changing dimensions but for data warehousing in general.

Lookup

Simply populating the data warehouse with data using ETL routines can be a time-consuming business, so to speed things up, the performance of the Lookup component has been significantly improved. It also has a new, easier to use user interface (UI), as shown in Figure 7-1.

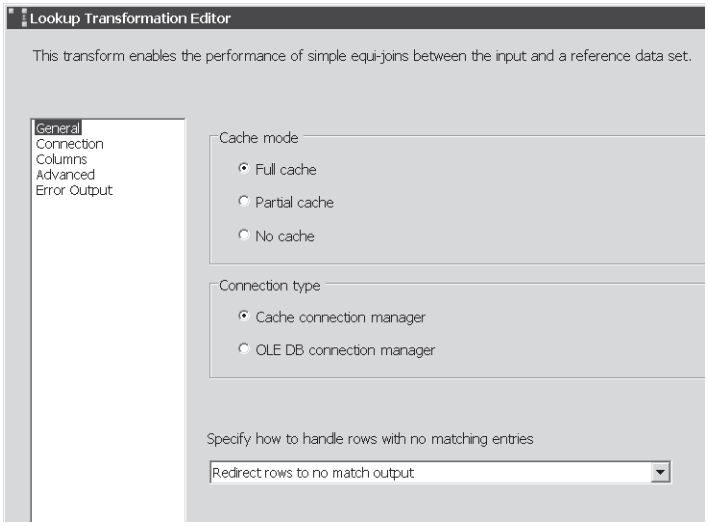


FIGURE 7-1 Lookup Transformation Editor dialog box

A Lookup is frequently used during the ETL process. Imagine that you have a fact table that has a column called EmployeeID. You also have a dimension table that lists the complete set of employees. A Lookup could be used to verify that all of the facts actually point to existing employees. In this case, the fact table is providing the source data, and the dimension table would be providing the reference set for the Lookup. (Lookups can, of course, do far more than simply verify data; they can also insert data from the reference set, such as the employee's name, into the source.)

The reference data can be directly addressed but, for reasons of efficiency, is often cached. In SQL Server 2008, the cache can be populated from a dataflow rather than from a SQL statement. (In SQL Server 2005, a Lookup could only use data from specific Object Linking and Embedding Database, or OLE DB, connections and be populated only by a SQL query.) The cache can now take data from many different sources: text, Excel, XML, Web services, or anything accessible using an ADO.NET provider. The Cache Connection Manager indicates the cache that is to be populated by the new Cache Transform component.

Another major improvement is that the cache no longer has to be reloaded every time it is used. If, say, one package has two pipelines that need the same reference dataset, the cache can be saved to virtual memory or permanent file storage, rendering it accessible to multiple Lookup components within a single package. The file format for the cache is optimized for speed and can be accessed many times faster than reloading the cache from a relational source.

Lookup has also gained a miss-cache feature. If the Lookup component is to run directly against the database, this feature can be set to include in the cache those key values from the source with no matching entries in the reference dataset. If the Lookup knows it has no value

of 999 in its reference dataset and that value comes in from the source, the component won't waste time and effort looking for it in the reference set again. It is reported that, in certain circumstances, this feature alone can improve performance by 40 percent.

Furthermore, no longer do miss-cache rows end up in the error output; there's a new "Lookup no match output" to which such rows can be directed.

Data Profiling

Before you can even think about writing a data cleansing process in SSIS, you have to understand the distribution of the data you are trying to clean. In the past, this has involved writing multiple GROUP BY queries, but now you can get a much better look using the new Data Profiling Task in SSIS. If you want to see it in action, try the following.

1. Create a new Integration Services project in Visual Studio 2008.
2. On the Data Flow tab, click the text that begins No Data Flow Tasks.
3. From the toolbox, drag an ADO.NET source onto the designer surface.
4. Click off the source to deselect it, and then double-click it.
5. Using the dialog box that opens, make a new connection to *AdventureWorks* (or the database of your choice), as shown in Figure 7-2.

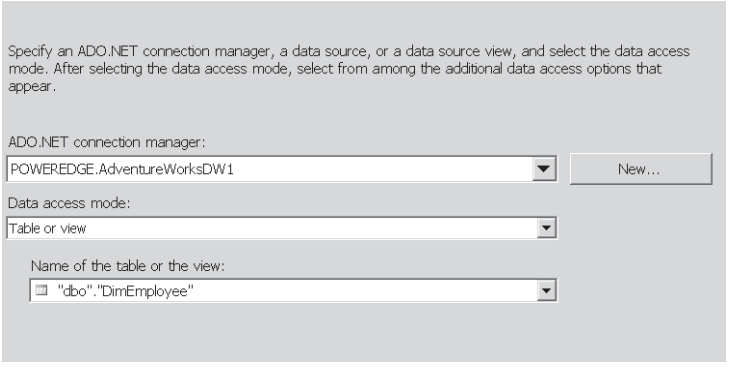


FIGURE 7-2 Connection dialog box

6. Click the Control Flow tab, where the new Data Flow Task should be visible.
7. Using the toolbox, drag and drop a Data Profiling Task from the list of Control Flow Tasks.
8. Click the Data Flow Task, and then click the green flow arrow, dragging until it connects the two.
9. Double-click the Data Profiling Task, and set up a file for the output. (Where you can put the file depends on your security settings: This example uses the desktop as a

convenient location.) Use the .xml extension, as this is the form in which the profile is output.

10. Click Quick Profile, and then choose the connection and the table you want profiled, as shown in Figure 7-3. Accept the remaining defaults. (You can, of course, play with these, but the defaults are just fine for seeing the profiler in action.)

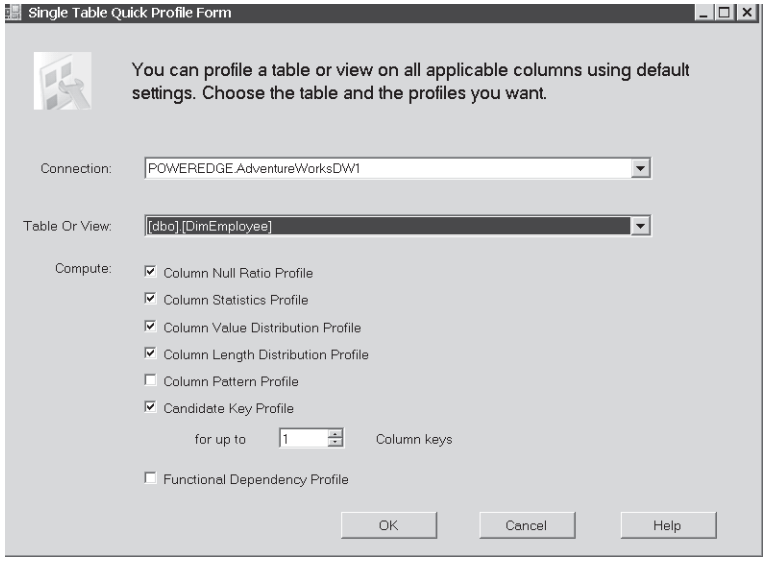


FIGURE 7-3 Single Table Quick Profile Form dialog box

11. Click OK. The profile properties are displayed. Click OK again. As shown in Figure 7-4, the two tasks are set up (their warning icon is no longer visible), and you're ready to go.

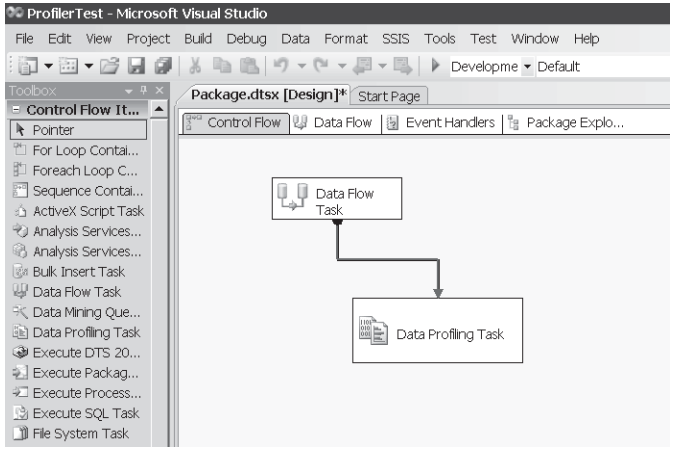


FIGURE 7-4 Completed Data Flow Task

12. Click the Start Debugging button (the one with a green button). The tasks change color on screen, and the Error List shows no problems—a successful outcome.

To inspect the profile itself, run DataProfileViewer.exe. You may find it at

C:\Program Files\Microsoft SQL Server\100\DTS\Binn\DataProfileViewer.exe

or

C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\DataProfileViewer.exe

Once you've found the program, run it and navigate to the location of the saved profiler .xml file. Expand the view to inspect the profile. As shown in Figure 7-5, you can see the Column Value Distribution Profile for the Title field. As you'd expect, there is but one CEO and many Buyers.

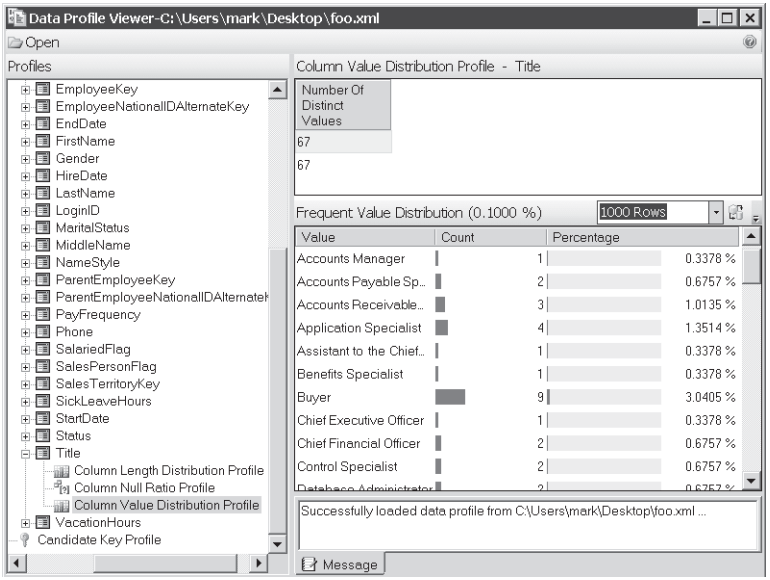


FIGURE 7-5 Data Profile Viewer dialog box

Apart from column value distribution, it also displays the Column Null Ratio Profile and Column Length Distribution Profile for each column. The Data Profiling Task is a cool tool and helps you get a feel for data before you embark upon cleaning tasks.

Other New Features

Scripting has improved so that you can use, for example, C# as a scripting language as well as VB.NET.

The SQL Server Import/Export Wizard (which generates SSIS packages) has been improved, as has the ADO.NET support.

Change Data Capture (CDC) means that source systems based on SQL Server 2008 will be able to capture changes to tables (inserts, updates, and deletes) and thus be able to work far more efficiently with SSIS.

SQL Server Reporting Services

In the past, to generate a meaningful, accurate report, you needed to understand both the data structure and the business context in which the data was to be used. This dual requirement has often generated a degree of tension in the delicate balance between IT professionals and business users. Each understood only half of the equation, so who got to author the report?

In SQL Server 2005, the IT professionals were provided with a tool called Report Designer in SQL Server Business Intelligence Development Studio, which allowed them to create reports. In addition, to help the business users create their own reports, Microsoft introduced the concept of a Report Model, which was created by the IT guys. It shielded the business users from the complexity of the online transaction processing (OLTP) or data warehouse environment yet provided them with access to the data they needed to create reports. The business users created the reports based on the model by means of a tool called Report Builder.

This arrangement was good for most users, but it tended to restrict the features available to power users on the business side. So, in SQL Server 2008, this has been addressed. Report Designer has been upgraded for developers, and it still ships as part of the SQL Server Business Intelligence (BI) Development Studio. For business users, Report Builder is now an office-optimized authoring environment that ships as a stand-alone tool. It is a more powerful design tool than Report Builder 2005, sharing many design components with Report Designer; in addition, it can now connect to many different data sources.

Report Designer in SQL Server Business Intelligence Development Studio

To use Report Designer, simply open SQL Server Business Intelligence Development Studio, and create a new Report Server project. Add a data source, right-click Reports in the Solution Explorer, and choose Add New Report, as shown in Figure 7-6.

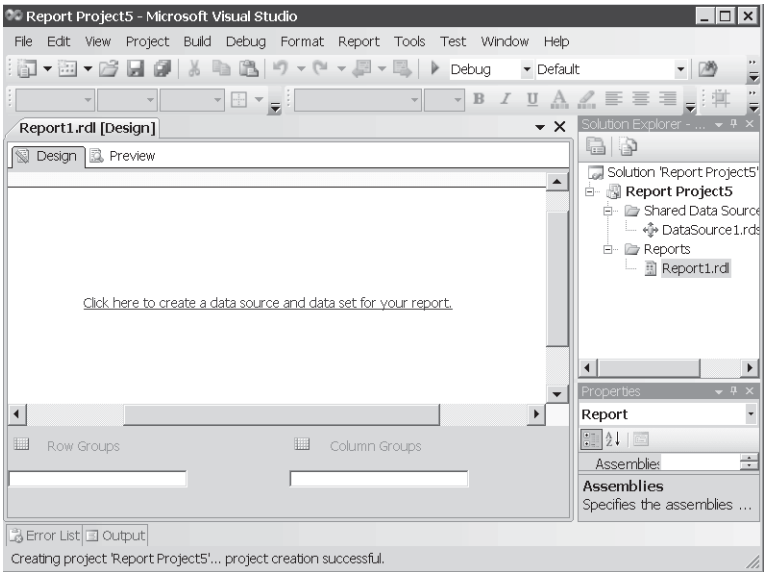


FIGURE 7-6 Report Designer

In Community Technology Preview 6 (CTP6), you click Here, as requested; in later CTPs and the final product, you simply see an empty report at this point. You then fill in the details, and, after clicking Next, you write a SELECT statement to pull in the data you need. In Figure 7-7, I am simply using a SELECT * against a view I created earlier.

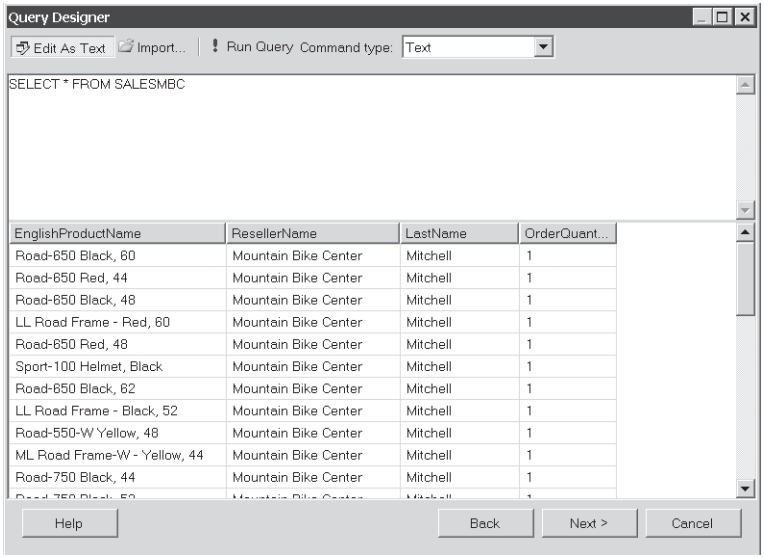


FIGURE 7-7 Query Designer dialog box

Finish off the dialog box requests, and you are ready to go.

Report Builder

Figure 7-8 shows Report Builder, the stand-alone report authoring environment for the business user. It has a ribbon interface to provide UI compatibility with the 2007 Microsoft Office System.

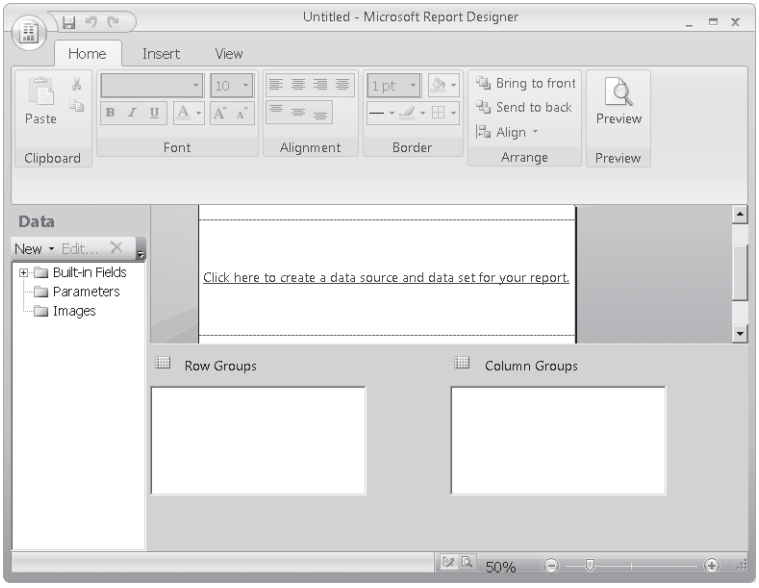


FIGURE 7-8 Report Builder 2008

Note In the current CTP (CTP6), Report Builder is called the Report Designer Preview and is located in:

C:\Program Files\Microsoft SQL Server\100\Tools\Reporting Services\ReportDesigner\ReportDesigner.exe

or:

C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Reporting Services\ReportDesigner\ReportDesigner.exe

In later CTPs, it will be renamed Report Builder 2008.

You set up the connection to the data source in much the same way as Report Designer. The two products share many components, such as the Layout surface, dialog boxes, data pane, and grouping pane.

New Controls in Both Authoring Environments

Once you have set up the data connection in either version, the two versions can be driven in very similar ways. Both allow you to access the new controls that have appeared, so let's take a look at those.

Tablix Data Region

SQL Server Reporting Services (SSRS) 2008 has a new Tablix data region as part of its Report Definition Language (RDL). It combines three existing data region types—Table, Matrix, and List—and brings together the features of all three. It supports them as layout types called (unsurprisingly) Table, Matrix, and List. These layouts form basic displays that you can then tweak by sorting, filtering, and calculating aggregate values. The new Tablix data region gives you considerably more control over layout and, specifically, lets you add column groups to a table layout and adjacent column groups to a matrix layout, in addition to static rows and columns for things like headers, labels, and subtotals.

However, if you look in the toolbox, as shown in Figure 7-9,

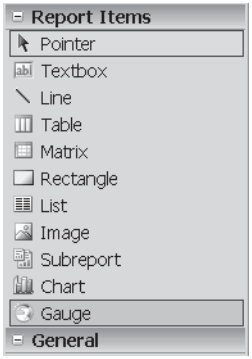


FIGURE 7-9 Report Items toolbox

you won't find a Tablix control as such—just Table, Matrix, and List. That's by design, and by dragging one onto the report, you can take a look at its properties, as shown in Figure 7-10.

Tables, Matrices, and Lists all share the same properties, so they really are the same—a Tablix control. Essentially, the difference between them is in the default structures of each.

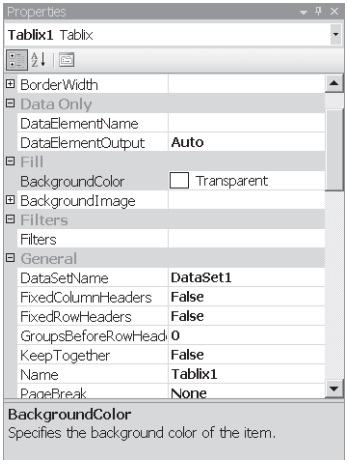


FIGURE 7-10 Tablix Properties dialog box

Gauge

We all knew that we wanted them, and here they are. A gauge is designed to display a single value or a small fixed set of values like target, value, goal, etc. The main difference between the chart and the gauge is that, with a chart, the number of data points may be unknown at design time and the query results determine the actual number. With a gauge, you know the number of data points in advance.

A bank of gauges is especially good for easy-to-read displays that compare several values: A group of gauges could, for instance, be used to display key performance indicators. You can also embed a gauge into a table or matrix data region and display data in linear or radial form.

Gauges allow you to present numerical data in a user-friendly fashion. I'll demonstrate this in the Visual Studio version of Report Designer, but it works pretty much the same in the stand-alone version.

From the toolbox, drag a gauge onto the report whereupon a box opens and shows the range of new toys that are available, as shown in Figure 7-11.

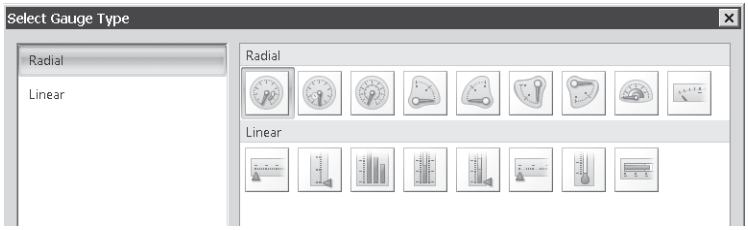


FIGURE 7-11 Gauge selection dialog box

I'll go for the 180 degree with range and put it onto the report, as shown in Figure 7-12.

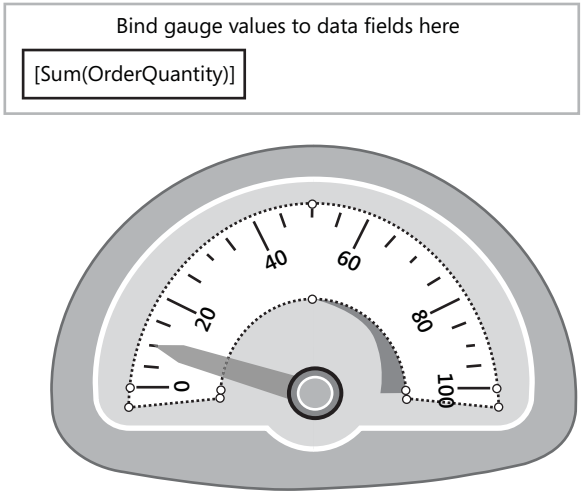


FIGURE 7-12 Radial pointer gauge

Clicking on the RadialPointer1 box allows you to choose a value that the gauge will show—you can even add the text version to prove that it's working, as shown in Figure 7-13.

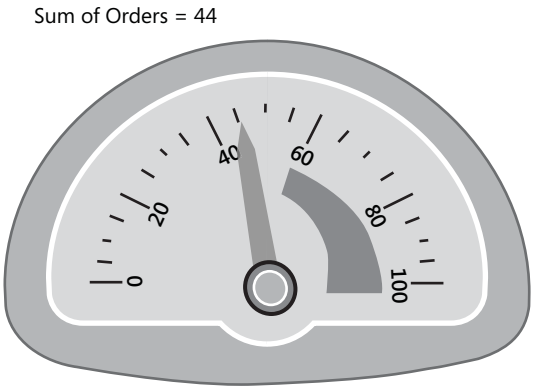


FIGURE 7-13 Radial pointer gauge with data

The gauge, like all of these controls, is highly configurable.

Chart Data Region

This data region has gained new visualization features: an extended list of chart types is available, now including polar, range, and shape. Figure 7-14 shows only a subset of those that are available.

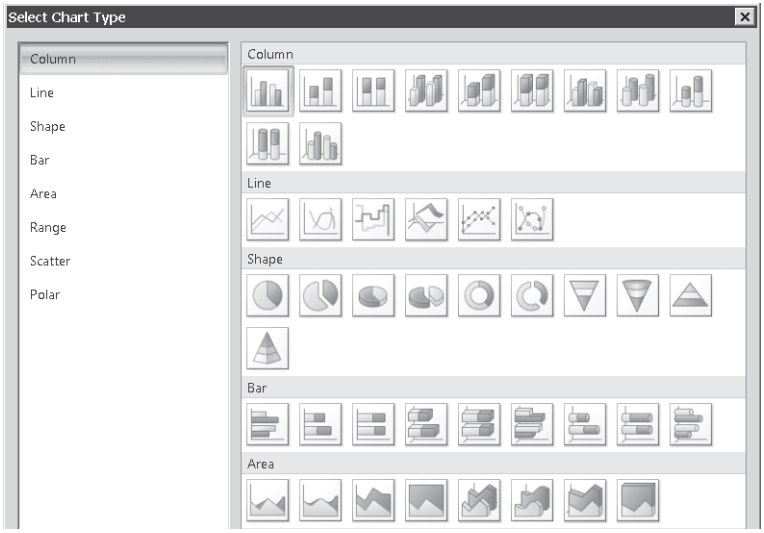


FIGURE 7-14 Select Chart Type dialog box

Secondary category and value axes have been introduced so that multiple data series can be displayed on more than one axis. For each axis, you have control over scale breaks, custom axis intervals, and interlaced strip lines. There is also support for run-time formulae.

Microsoft Office Rendering

This is a new feature that lets users of Word access reports. The report is rendered as a Word document compatible with Word versions back to and including Word 2000. This is a great addition that provides an important method of disseminating report data throughout an organization, putting the information within easy reach of a much larger portion of the workforce.

The existing Excel renderer has been enhanced and now supports features such as nested data regions, merged cells, and sub-reports.

These renderer enhancements let users gain the insights into data that a good report can offer while working with familiar Office tools.

Finally, coming in the next CTP refresh, Reporting Services will provide support for rich text. This will enable the developer to have much finer control over the report formatting and, hence, the creation of much more sophisticated reports.

SQL Server Analysis Services

One of the main enhancements in SQL Server 2008 Analysis Services (SSAS) is focused on speeding up Multidimensional Expressions (MDX) queries—particularly those that potentially touch a large number of null values. The mechanism that produces this performance enhancement is known as block computation. In addition, there have been huge improvements in terms of the backup performance. Finally, there is the new Scalable Shared Database (SSD) feature, which allows you to scale your Online Analytical Processing (OLAP) query workload across many servers with just a single copy of the database.

Block Computation

Data in the cube space is often sparse, which is another way of saying that it is often full of null values. Think about it this way: A fact table stores information about events that actually occur—only when a customer buys a product from an employee is a row entered. In one week, a busy store might have 10,000 facts.

ProductKey	TimeKey	EmployeeKey	CustomerKey
372	762	774	769
287	762	234	445
380	762	124	8765
390	762	565	5

A multidimensional matrix, on the other hand, theoretically stores all of the possible intersections for Time, Employee, Customer, and Date. Given 10,000 customers, 100 employees, 1,000 products, and five days, that's potentially just less than 5 billion nulls. In practice, of course, data compression reduces this extensively, but the nulls are still there in the sense that if you ask to see the value, a null has to be returned. It turns out that these nulls, if present in large numbers, can significantly slow queries.

Block computation solves this. Of course, the internal structure of a cube is very complex, so any description of how internal processes work is necessarily an abstraction, but you can picture the improvements that block computation brings in the following way.

Imagine an MDX calculation that calculates a running total for two consecutive years, something like this:

```
CREATE MEMBER CURRENTCUBE.[Measures].RM
   AS ([Order Date].[Hierarchy].PrevMember,[Measures].[Order Quantity])+
      Measures.[Order Quantity],
FORMAT_STRING = "Standard",
VISIBLE = 2 ;
```

Further imagine that there are very few values for most products in the years 2003 and 2004. The following tables, showing a small subset of products, illustrate this.

	2003	2004
Product 1		
Product 2	2	
Product 3		
Product 4		5
Product 5		
Product 6		
Product 7	3	1
Product 8		
Product 9		

The following query will return the calculated measure RM for all products ordered in 2004.

```
SELECT [Order Date].[Hierarchy].[all].[2004] on columns,
[Dim Product].[Product Key].[All].children on rows
From Foo
Where Measures.RM
```

Let's look at how this question is solved in SSAS 2005. The evaluation proceeds one cell at a time. For each cell, for the current year (2004, in this case), go to the previous year, pick up the quantity ordered, and add it to the quantity ordered in the current year. This approach has two major speed disadvantages.

First, the step of navigating to the previous year to see if data is present is performed as many times as there are products. You are, in effect, ascertaining that data exists for 2003 potentially thousands of times. The nature of a hierarchy tells you that, if there is order quantity data present at the intersection of *any* product for 2003, it should be present for *all* products. This one-cell-at-a-time approach involves huge duplication of effort and will inevitably slow query performance, and it is a problem regardless of whether there are many nulls or none. It's the act of checking to see if the data exists that takes the time and effort.

Second, the data from each pair of cells for each product are calculated. Where there are many nulls, a high proportion of those calculations return a null, and every single one of those calculations represents wasted effort. Looking at the preceding tables, the only products for which you need to calculate the value are 2, 4, and 7. Where there are many nulls, the effect on performance can be devastating.

So what does SSAS 2008 do to improve query speed? The whole approach to processing the query has changed, and it addresses both the problems outlined above.

The "many checks for previous period data" problem is fixed by a straightforward technique. Data for the current year is pulled back and then a single check is made to see if data exists for the previous year. It sounds simple, it is simple, and it improves performance dramatically. If data exists, it is pulled back.

The "making many pointless calculations" problem is solved in an equally straightforward fashion. Before any calculations take place, the order quantity data for 2003 and 2004 is inspected, and any cells containing null values are removed.

	2003
Product 2	2
Product 7	3

	2004
Product 4	5
Product 7	1

Once this has been done, the two sets of data can be compared:

	2003	2004
Product 2	2	
Product 4		5
Product 7	3	1

The calculations used only data with the potential to contribute to a meaningful result. In other words, calculations are only performed for products with a value in both years or with a value in either one of them.

Block computation can improve the speed of querying by several orders of magnitude, given the right (or should that be wrong?!) distribution of data.

Analysis Services Enhanced Backup

There is a new backup storage subsystem in SSAS 2008, shown in Figure 7-15. The bad news about this feature is that, just like block computation, there is nothing you can do with it to show off your prowess; it just works. The good news is that you simply get far, far better backup performance without having to change a single line of code. So the good news

outweighs the bad by several orders of magnitude. In addition, the backup now scales much more effectively, so it can handle SSAS databases of well over a terabyte in size. For many large volume users, this means they can stop doing raw file system copying of data files and use the backup system.

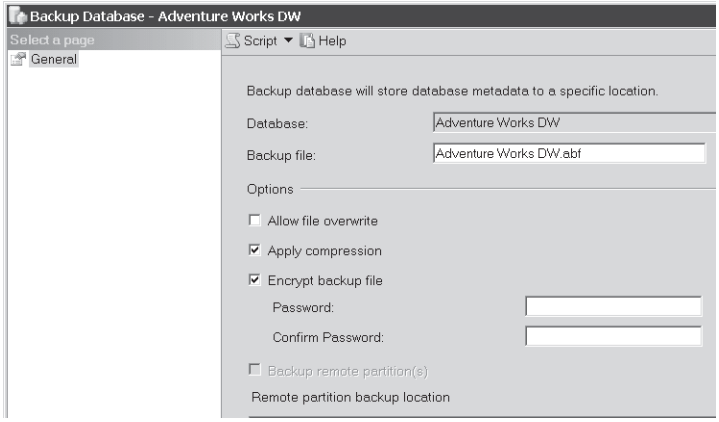


FIGURE 7-15 Backup Database dialog box

A side effect of this new subsystem is that the format of the SSAS backup files has changed (although the file extension remains the same). While the new subsystem won't save in the 2005 format, it is fully backward compatible with the old format, which means you can restore databases in SSAS 2008 that were backed up in SSAS 2005.

Enhancement to Writeback Performance

It has long been possible in SSAS to write back cell values (at both leaf and aggregation level) to the cube. In practice, these values are written to a special Relational Online Analytical Processing (ROLAP) writeback partition, which stores not the new value but the difference (the delta) between the old and the new values. This means the original value can still be retrieved from the cube, and, when an MDX query asks for the new value, both partitions are queried and the aggregated value (the original summed with the delta) is returned.

In SSAS 2005, these writeback partitions had to be ROLAP (stored in relational tables), which introduced a potential bottleneck in the performance. By now, you are already way ahead of me on this enhancement. Yes, in SSAS 2008, these writeback partitions can be stored as Multidimensional Online Analytical Processing (MOLAP). As it happens, this can slow down the writeback commit fractionally (because the engine has to update the MOLAP partition data as well as the writeback table), but that is very rarely a bottleneck in practice. The huge gain is that query performance is dramatically improved in almost all cases.

Scalable Shared Databases for SSAS

As SQL Server BI becomes more pervasive, the number of users requiring access to cubes is increasing. There are two possible solutions, scale up or scale out.

Scale up is conceptually very simple—you buy a very large server with multiple processors and lots of RAM, and you place the cube on that. If you use a large enough server, this solves the problem and has the added advantage that individual query performance is significantly enhanced. Unfortunately, so is the cost: Large servers tend to cost more per CPU than multiple small servers, which can be very inexpensive.

SSAS 2005 provided a scaleout solution in the sense that you could replicate the data across multiple servers and then put a load balancing solution such as Microsoft Network Load Balancing (NLB) between the servers and the users. This could be (and is) made to work, but there is a significant overhead in set up and maintenance.

SSAS 2008 introduces a new scaleout solution feature, which is called Scalable Shared Database (SSD), not unreasonably because it works in a very similar way to the SSD feature introduced in the SQL Server 2005 relational database. This consists of three logical components:

- Read-only database, which allows a database to be marked as read-only
- Database storage location, which allows a database to reside outside the server Data folder
- Attach/detach database, which allows the database to be attached or detached from any Universal Naming Convention (UNC) path

These three features can be used independently to your advantage, but when used together, they enable a scale-out solution to be constructed relatively easily.

Other New Features

- Check out the Dynamic Management Views (DMV) in Analysis Services 2008 and also the resource monitoring. MDX has a whole string of enhancements in addition to block computation, such as dynamic sets, non-visual subselects, and calc members subselects.
- Analysis Management Objects warnings: SSAS 2008 now shows real-time suggestions/warnings about design and best practice as you work. They display in the UI as blue wiggly underlines, and hovering over the underlined object will display the warning.

Business Intelligence Design Studio (BIDS) also has some major enhancements, such as the attribute relationship designer, the aggregation manager, and the usage-based optimization.

Summary

There is a huge list of improvements and new features in SQL Server 2008:

- MERGE
- Lookups
- Profiling
- Report development improvements
- Tablix
- Gauge
- Office rendering
- Rich text support
- Block computation
- Enhanced backup
- Enhancement to writeback performance
- Scalable Shared Databases for SSAS

And the list goes on. These are all features that are designed to make life easier, either for the BI developer or the end user. Without a doubt, they add up to an excellent reason to upgrade to SQL Server 2008.

For more detail on the new features in SQL Server 2008, check out Microsoft's white papers on the subject; for example: *http://www.microsoft.com/sql/techinfo/whitepapers/SQL2008IntroDW.mspx*.

Two other white papers titled "Best Practices for Scale up Data Warehousing in SQL Server 2008" and "Best Practices for Data Warehousing with SQL Server 2008" are in preparation as we finish this book and should be available from the Microsoft Web site.

Index

F

G

H

I

L

M

T

U

V

W

X

Peter DeBetta

Peter DeBetta, a Microsoft MVP for SQL Server, is an independent consultant, author, and architect specializing in design, development, implementation, and deployment of Microsoft® SQL Server, Microsoft SharePoint Server, and .NET solutions.

Peter develops software, teaches courses, presents at various conferences, including Microsoft TechEd, PASS Community Summit, VSLive!, DevWeek, DevTeach, and SQLConnections. Peter writes courseware, articles, books (like this one), and is a co-founder of the popular *SQLblog.com* website.

In his spare time, you can find Peter singing and playing guitar, taking pictures, or simply enjoying life with his wife, son, and daughter.

Greg Low

Greg is an internationally recognized consultant, developer, and trainer. He has been working in development since 1978, holds a PhD in Computer Science and a host of Microsoft certifications. Greg is the country lead for Solid Quality, a SQL Server MVP, and one of only three Microsoft Regional Directors for Australia. Greg also hosts the SQL Down Under podcast (*www.sqldownunder.com*), organizes the SQL Down Under Code Camp, and co-organizes CodeCampOz. Greg is a Professional Association for SQL Server (PASS) board member.

Mark Whitehorn

Dr. Mark Whitehorn specializes in the areas of data analysis, data modeling, data warehousing, and business intelligence (BI). He works as a consultant for a number of national and international companies, designing databases and BI systems, and is a mentor with Solid Quality Mentors.

In addition to his consultancy practice, he is a well-recognized commentator on the computer world, publishing about 150,000 words a year, which appear in the form of articles, white papers, and books. His database column in PCW has been running for 15 years, his BI column in Server Management magazine for five.

For relaxation he collects, restores, and races historic cars.